# A QUERY LANGUAGE
FOR THE RELATIONAL DATA BASE

By

V I N O D   G U P T A

COMPUTER SCIENCE PROGRAMME
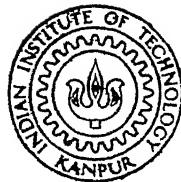
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

JULY, 1980

A QUERY LANGUAGE
FOR THE RELATIONAL DATA BASE

A Thesis Submitted
In Partial Fulfilment of the Requirements
for the Degree of
MASTER OF TECHNOLOGY

By
VINOD GUPTA

to the
COMPUTER SCIENCE PROGRAMME
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR
JULY, 1980

CSP-1980-M-GUP-QUE

<center>CERTIFICATE</center>

This is certified that the work entitled, 'A QUERY LANGUAGE FOR THE RELATIONAL DATA BASE' has been carried out under my supervision by Shri Vinod Gupta and it has not been submitted elsewhere for a degree.

(V. Rajaraman)
Professor of Computer Sciences
Indian Institute of Technology,
Kanpur

July 11, 1980

ACKNOWLEDGEMENTS

I wish to express my gratitude to Prof.V.Rajaraman for his inspiring guidance throughout this work.

Thanks are also due to Dr. B. Srinivasan for the fruitful discussions I have had with him.

Finally, I must thank Mr. J.K. Misra for his expert typing work.

Kanpur
July, 1980   Vinod Gupta

CONTENTS

FIGURES AND LISTINGS

ABSTRACT

This thesis suggests a new 'Query Language' for information retrieval from a Relational Data Model. It also shows the implementability of the language. All the system programs have been written in PASCAL, because of the excellent character manipulation capabilities of the language. The system has been successfully implemented on DEC 1090 System available in the Computer Centre of Indian Institute of Technology, Kanpur.

CHAPTER J

INTRODUCTION

1.1 Data Base Management System:

DATA, as usually conceived of, is some collection of
information, gathered from somewhere, with some intentions –
may be clearly defined but not usually so. If allowed to
remain in its crudest form, the data is of no practical use
to anyone, even if the purpose for which it was collected was
well defined. It has to be converted to a form such that it
can be comprehended, partly if not wholly, by anybody who
wants to make use of it. By converting it to some such form
and storing it, may be in the secondary memory of a computer,
one is moving towards, what we call as 'building a data base'.
But a lot more needs to be done. How to get the desired
information out of this stored structure like magnetic disk or
tape? Some methodology has to be adopted for the purpose.
This methodology, technically known as 'Data Base Management
System', should be such that a user can construct his own
'information need statements' and interrogate the data base
without having much knowledge of the system. Of course, he
is expected to know what data is stored in the system which
he wishes to interrogate. (Otherwise what is he looking for
anyway).

The user of a data base could be anyone. For example, in an industrial organization, he could be anyone right from the foreman at the shop floor level to the top managers of the organization. Each of these persons has a different way of thinking and expressing his needs. Whereas a computer system, like the DBMS, requires constructs in a very formal language before it can cater to the needs of the user.

1.2 Motivation:

In this thesis, an attempt has been made at describing a new language which enables one to construct queries to interrogate a Relational Data Base. The statements in the language are such that, with the knowledge of the syntax of the language, they come naturally from the word statement of the query. The idea is taken from 'Decision Tables' because it is felt that any query can be naturally split into two parts - a condition part and an action part. Moreover, the concept of a decision table is easy to understand.

A Relational model has been chosen for the data base because of its undoubted conceptual superiority over the other models like the Network model and the Hierarchical model. It is assumed that the relations stored in the data base are in the Third Normal Form.

The motivation behind this work has come from the fact that, in general, the user of any data base will not be a computer professional. Any occasional user (of a probably

large data base) must be able to satisfy his information
need without having to hire a computer professional for
the purpose. In other words, this thesis suggests a Higher
Level Language with the idea that the user will find it easy
to program his queries in this language. It also makes an
attempt at showing that the language is complete and it
can be implemented on any of the present day big computer
systems. Examples have been taken from Codd's paper [4]
to show that the language can retrieve information for all
sorts of complex but meaningful queries.

1.3 Organization of the Thesis:

After a brief introduction in Chapter I, a study
of the present relational data base systems has been done
in Chapter II. Chapter III describes the new proposed
language. It shows how the different constructs have been
developed and gives the full syntax details of the language.
Chapter IV is devoted to the implementation details, which
include processing of queries in the language and converting
them to the relational algebra expressions. Algorithms for
implementing the different relational algebra operations
have been given in Chapter V. The concluding chapter
summarizes the work done and indicates the scope for
improving the system.

CHAPTER II

PRESENT STATE OF ART

2.1 The 'Relational Model':

The relational model of data base was introduced by
E.F. Codd of the IBM Research Laboratories in 1970. He was
particularly interested in providing the user of the data
with a data model which would be independent of the various
implementation considerations. He was also interested in
making available to the user a high-level language, which
would be non-procedural in nature, for accessing the data.
In his first paper (ref. [6]) he says that,

'The relational model of data provides a means of
describing data with its natural structure only - that is
without superimposing any additional structures for machine
representation purposes. Accordingly, it provides a basis
for high-level data language which will yield maximal indepen-
dence between programs on the one hand and machine representa-
tion and organization on the other. A further advantage of th
relational view is that it forms a sound basis for treating
derivability, redundancy and consistency of relations'.

A relation is defined in mathematics as follows: Given
sets D1, D2, ... Dn (not necessarily distinct), R is a relatio
on these n sets if it is a set of ordered n-tuples d1,d2,...dn
such that d1 belongs to D1, d2 belongs to D2, and so on.

Sets D1, D2,...Dn are called the domains of R. The value n
is called the degree of R.

It is convenient to represent a relation in the form
of a table, where each row represents one n-tuple. The relation
'Part' shown in Figure 2.1 consists of 6 tuples.

Fig. 2.1: A typical relation.

PART

P #	PNAME	COLOR	WEIGHT	QOH
1	NUT	RED	12	4
2	BOLT	GREEN	17	7
3	SCREW	BLUE	17	3
4	SCREW	RED	14	9
5	CAM	BLUE	12	2
6	COG	RED	19	1

It is observed that a relation has the following
properties:

(1)   No two rows are identical,

(2)   Ordering of rows is insignificant,

(3)   Ordering of columns is also insignificant,

(4)   Every value within a relation is an atomic data item.

The last property is true in case the relation is in the
'Third Normal Form' (Ref. [5]).

It was pointed out by Codd that the relational operations
defined in mathematics can be applied on these relations. He

showed how to carry out the operations like projection and join on these relations.

Most existing systems are not relational as a natural outcome of the way in which computing technology has itself developed. The earlier computer systems had comparatively small capacity and high access time. Traditionally, the emphasis was on sequential media like tapes and cards. With modern hardware and techniques, it has become possible to design and implement more large scale systems based on the relational approach.

2.2 Data Sub Languages for the Relational Model:

For query purposes in a relational model, a data sub language (DSL) should allow the user to specify the relation he wants retrieved from the data base. It is to be pointed out here that any information that a user obtains from the data base will be in the form of a new relation which will k derived from the original relations stored in the system. There are two ways in which a user can specify the new relation he wants retrieved from the data base —

(1) A non-procedural method, in which he would simply state the definition of the desired result and leave it to the system to determine the required operations.

(2) A procedural method, in which he would actually spec: the step-by-step operations to be performed to produce the

desired result. (The operations here refer to the relational operations - join, project and divid·).

2.2.1 Codd's DSL ALPHA is a non procedural (calculus based) language. It is data independent in that it contains absolutely no references to storage/access details such as indexes, pointers etc. The language is complete. The complexity of a statement in DSL ALPHA is in direct proportion to the complexity of the operation the user is trying to perform.

2.2.2 A procedural language (algebra based) also provides full data independence. It is a reasonably simple language but probably less simple than DSL ALPHA. Compared to most existing languages, which tend to be very low level, the algebraic language can be called to be non-procedural. But compared to DSL ALPHA it is definitely more procedural.

2.2.3 As an illustration of the types of statements in the two languages mentioned above, let us take the following query based on the relations given in Fig. 3.1.

Word Statement of query:

    Find the part names of parts supplied by supplier 'Jones'.

The query as converted to one in relational algebra:

        JOIN SUPPLIER AND SUPPLY (over S#);

        JOIN the result and PART (over P#);

        PROJECT the result by PNAME where SNAME = 'Jones'.

The query as converted to one in relational calculus:

GET W P.PNAME : $\exists$ SUPPLY((PART.P# = SUPPLY.P#)

$\wedge$ (SUPPLY.S# = SUPPLIER.S#)

$\wedge$ (SUPPLIER.SNAME = 'JONES')).

For the sake of simplicity it is essential that the language made available to the user be non-procedural. The DSL ALPHA would be a good candidate for the same. But writing queries in DSL ALPHA requires the knowledge of 'first order predicate calculus'. Symbols like '$\exists$' (there exists), '$\forall$' (for all) and ':' (such that) tend to scare away the user of this language.

2.3 Existing Systems:

There have been quite a lot of small relational systems in the recent past. But most of them do not insist on any particular level of normalization. They are content with the relations being in the First Normal Form. Obviously a data model in the 1NF can not exploit most of the potentialities of a relational data base.

2.3.1 MacAIMS appears to be the earliest example (1970-71) of a system providing both a full relational model and a language (based on algebra) which actually views the data in its relational form. The language, however, is not complete, in the sense that not all relations derivable from the data model by means of first order predicate can be retrieved in this language. One interesting feature about

MacAIMS is that the structure may vary from relation to relation
(thus allowing each relation to be stored in the form most
suited to it). For details of the language refer to [8].

2.3.2 MORIS (1972) incorporates some of the ideas of
MacAIMS. In it, however, the user's view of the data may
include unnormalized relations, which are defined in terms
of the fundamental normalized relations in the data model.
Thus the user is provided with a limited form of data sub-
model. Though the language is based on relational calculus,
yet it is (also) possible to perform a retrieval using
algebraic operations. For further details refer [9].

2.3.3 The IS/1 system (1972) also provides a limited form
of data submodel. An 'implied relationship' is defined by
a set of (algebraic) statements of this language which
specify how other relations are to joined, projected, and
otherwise manipulated to produce the required result. The
first time the user refers to this relation, the definition
is invoked and the desired relation is generated. Subsequent
operations are performed directly on the generated relations.
This system developed some useful optimization techniques
for evaluating algebraic expressions. For further details
of this language refer to [10].

2.3.4 The SEQUEL prototype was originally developed as a
single user system to demonstrate the feasibility of support-
ing the SEQUEL language (1974). This language is complete in

the sense that any relation, derivable from the data model
by means of first order predicate, can be derived in this
language. It does not involve symbolism of the predicate
calculus (i.e., there are no quantifiers). It is based
on a construct known as 'mapping' which may be viewed as a
subsetting followed by a projection. Its syntax is quite
English-like. As an example the query above may be written as

      SELECT PNAME FROM PART

      WHERE P # =

            SELECT P # FROM SUPPLY

            WHERE S # =

                  SELECT S # FROM SUPPLIER

                  WHERE SNAME = 'JONES'.

However, this system has been extended by the IBM Cambridge
Scientific Centre and the MIT Solan School Energy Laboratory
to allow a single type of concurrency and is being used as a
component of the Generalized Management Information System
(GMIS) being developed at MIT for energy related applications
For further details on the language refer to [7] and [11].

Of the systems given above, MacAIMS is not complete
and MORIS and IS/1 present to the user a data submodel which
has unnormalized relations. SEQUEL is a language which is
complete and without quantifiers. But the ideas of 'mapping'
and 'recursion', which are made use of in this language, are
not very simple to grasp by an ordinary user. Most of the

languages of the type SEQUEL, though avoid the use of quanti-
fiers of the relational calculus, require the user to evolve
a method of generating the output relation. Like in the
above example, where the constraint is on the supplier name
and the attribute required in the output relation is partname,
the SEQUEL query clearly specifies how the 'SUPPLY' relation
is to be used to reach the 'PART' relation starting from the
'SUPPLIER' relation.

The language we are looking for should be reasonably
complete (equivalent to DSL ALPHA) and should also not make
use of any quantifiers. Apart from this, it should not
involve any difficult concepts. As in the above example,
we should not expect the user to know how the path is to
be traced while executing his query. If need be, he should
be expected to know only this much that the relation 'SUPPLY'
will be involved in getting for him the output relation he
requires, and not how it has to be used.

CHAPTER III

FORMULATION OF LANGUAGE

The basic idea behind specifying this language is that a query can be split into a condition part and an action part. Then it should be possible to enumerate all the possible types of conditions and actions. Finally it should be possible to combine those types of conditions and actions which require the same type of manipulations on the relations.

3.1 Parts of a Query:

First we show that any query can be split into a condition and action. Then we go on to studying these two parts separately. Let us start with considering a few examples (refer to the relations in Fig. 3.1).

Query Find the part name and quantity-on-hand of parts whose quantity-on-hand is less than 5.

Clearly the condition on all tuples in the output relation is that the quantity-on-hand for the part be less than 5. There is no other constraint that one can obviously see. The action part is that it is required to get into the workspace the part names and the quantities-on-hand. Thus the query can be written as -

Supplier

S#	SNAME	LOC
1	SMITH	LONDON
2	JONES	PARIS
3	LAKE	PARIS
4	CLARK	LONDON
5	ADAMS	ATHENS

Supply

S#	P#	J#	QTY
1	1	1	2
1	1	4	7
2	3	1	4
2	3	2	2
2	3	3	2
2	3	4	5
2	3	5	6
2	3	6	4
2	3	7	8
2	5	2	1
3	3	1	2
3	4	2	5
3	6	3	3
3	6	7	3
5	2	2	2
5	2	4	1
5	5	5	5
5	5	7	1
5	6	2	2

Part

P#	PNAME	COLOR	WEIGHT	QOH
1	NUT	RED	12	4
2	BOLT	GREEN	17	7
3	SCREW	BLUE	17	3
4	SCREW	RED	14	9
5	CAM	BLUE	12	2
6	COG	RED	19	1

Job

J#	JNAME	MGR-E#
1	SORTER	24
2	PUNCH	11
3	READER	17
4	CONSOLE	9
5	COLLATOR	27
6	TERMINAL	18
7	TAPE	13

Figure 3.1 Relations In the Data Base

Condition: Quantity-on-hand of part less than 5.

Action: Get part name and quantity-on-hand into the
             work space.

Let us look at another example

Query   Find the names and locations of all suppliers, each of
        whom supplies all jobs.

The condition here is that all the jobs be supplied. By
all jobs it means that the jobs, listed in the relation JOB,
be treated as one entity.   The action required is to get into
the workspace the names and locations of suppliers. Thus the
query can be written as:

Condition: All jobs.

Action: Get into the workspace, name and location of
             suppliers.

Let us go on to look at another slightly complex query:

Query   Find the name and location of all suppliers who supply
        atleast those jobs supplied   by supplier 'SMITH'.

This query being slightly complicated, has to be split
into two smaller queries.

Sub Query A   Find the job numbers of jobs supplied by supplier
              'Smith',

Sub Query B   Find the name and location of suppliers who supply
              atleast those jobs whose job numbers are in the
              workspace above.

This two step query can be split into condition and action as follows:

Condition: Supplier name 'Smith'

Action: Get into workspace the job numbers supplied.

Condition: All job numbers in the above workspace.

Action: Get into the final workspace supplier names and locations.

From the three examples that we have considered, it seems that splitting a query into its condition and action components is not difficult.

3.2 Workspace:

It can be noted that the concept of workspace is important. Workspaces have the following features:

(1) They are not part of the data base itself, so that there will be no inconsistency due to the creation of redundant data.

(2) The work spaces b long to one individual user or process, so that the manipulations of data within such workspaces do not conflict with other users.

(3) A naming scheme is employed to distinguish relations and attributes within the work space from the relations and attributes in the data base.

Now we look at the condition part in more details.

3.3   Condition Part:

This is essentially the same as the predicate in
DSL ALPHA. Like a predicate, a condition may consist of an
expression of arbitrary complexity, formulated according to
the usual rules. The permitted operators are the comparison
operators '=', '< >', '>', '<', '$\geq$', '$\leq$', the Boolean operators
'OR', 'AND', and 'NOT' and of course parentheses '(' and ')'
to enforce a desired order of evaluation.

The condition part of any query can be reduced to a
normal form:

$$C = (C11 \text{ AND } C12 \text{ AND } \ldots \text{AND } C1N_1)$$
$$\text{OR } (C21 \text{ AND } C22 \ldots \text{ AND } C2N_2) \ldots$$
$$\text{OR } (CM1 \text{ AND } CM2 \ldots \text{ AND } CMN_M)$$

where Cij_m is a <u>simple condition</u> with one comparison operator.
The result of a general query, with the condition part as 'C',
can be obtained by applying 'union' and 'intersection' opera-
tions on the results of individual queries with simple condi-
tion parts. For example, the final result for condition 'C'
will be given by,

$$R = (R11 \cap R12 \ldots \cap R1N_1) \cup (R21 \cap R22 \ldots \cap R2N_2)$$
$$\ldots \cup (RM1 \cap RM2 \ldots \cap RMN_M),$$

where Rij_m is the relation retrieved by a query with the
(simple) condition part Cij_m. The system is designed for a
general condition of the above type. For explanation purposes
however, we will consider only queries with simple conditions.

Now let us enumerate the different types of possible conditions required to incorporate all types of queries. Here it is to be pointed out that effort has been made to include all types of conditions that are encountered in queries. But the system can be easily modified for including any further types that do not fall in any of the types that are described here. This will be evident from the structure of the system.

Given below is the list of seven different types of conditions. An example query has been given with each for explanatory purposes.

(1)   The value of an attribute in a relation is equal (or unequal) to a specified value. For example –

Query 1: Get the part name and quantity-on-hand of 3 parts
          for which the quantity-on-hand is less than 5.

(2)   The value of an attribute in a relation is equal (or unequal) to (a) the n-th largest value for that attribute in that relation; (b) the n-th smallest value for that attribute in that relation. For example –

Query 2: Find the part number and part name of parts for which
          quantity-on-hand is the largest.

(3)   Something is required to be done for each of the values of an attribute in a particular relation. For example –

Query 3: Find in the increasing order of quantity-on-hand,
          the part name and quantity-on-hand of parts being
          supplied.

(4) In a relation, the number of tuples, with same values for some attributes and different values for some other attributes in that relation, is equal (or unequal) to a specified number. For example --

Query 4: Get the number of parts being supplied to more than two jobs.

(5) In a relation the total of an attribute, with same values for some other attributes in that relation, is equal (or unequal) to a specified value. For example -

Query 5: For those parts, for which the total quantity being supplied to a job is greater than 5, get, as a triple, part number, the job number and the total quantity of that part being supplied to that job.

(6) All the different values of an attribute in a relation are to be treated as one entity. For example -

Query 6: Get the supplier names and locations of suppliers who supply to all jobs.

(7) The value of an attribute in one relation is not present in the values of that attribute in another relation. For example -

Query 7: Get the name and location of those suppliers who do not supply to any job.

We now construct statements of the language for these conditions. It can be noted that the comparison operator for

conditions of the types 1,2,4 and 5 could be any of the six, i.e., =, < >, <, >, <=, or >=. For conditions of the type 3 and 6, only the equality (=) operator is valid and for condition of the type 7, only the 'not equal to' (< >) operator is valid. The syntax for these simple conditions is as follows-

(1)   Relation name.attribute name <COND> " value".

The condition for example Query 1 becomes -

PART.QOH = " 5 "

(2)   (a) Relation name.attribute name <COND> n TOP

(b) Relation name.attribute name <COND> n BOTTOM

The condition for example Query 2 becomes -

PART.QOI = 1 TOP

(3)   Relation name.attribute name =   *

The condition for example Query 3 becomes -

SUPPLY. P # = *

(4)   COUNT (relation name SAME attribute names DIFFERENT
                attribute names) <COND> positive integer

The condition for example Query 4 becomes -

COUNT (SUPPLY SAME P #   DIFFERENT J #) > 2

(5)   TOTAL (relation name SAME attribute names:attribute
                name) < COND> value

The condition for example Query 5 becomes -

TOTAL (SUPPLY SAME P # , J # : QTY) > 5

(6)   Relation name.attribute name = ALL

The condition for example Query 6 becomes -

JOB.J # = ALL

(7)   Relation name.attribute name < > relation name.attribute name

The condition for example Query 7 becomes –

SUPPLIER.S# <> SUPPLY.S#

Here '<>' is the symbol used for 'not equal to'. The reserved words of the language are given in capital letters. <COND> refers to any of the six comparison operators.

We have considered only queries with a simple condition part. Now we give examples of two more queries which have a general condition part.

Query 8:   Get the names of those suppliers who supply to job number 4 and are located in Athens.

The condition for example query 8 becomes –

SUPPLIER.LOC = " ATHENS" AND SUPPLY.J#= " 4 "

Query 9:   Get the names and locations of suppliers who supply a blue part or a green part.

The condition for the example query 9 becomes –

PART.COLOR = " BLUE" OR PART.COLOR = " GREEN"

3.4   Action Part:

This part of the query just indicates the domains (i.e. attribute names) to be brought into the work space. It is analogous to the 'target list' of DSL ALPHA. Apart from speci-fying the attribute names (qualified by the appropriate relation names) to be retrieved, it might (also) be required to do the totalling of a certain field before displaying (example Query 5) or it might be required to get the count of the number

of tuples (that satisfy the condition) in the output relation
(Example Query 4). Sometimes one might want to retrieve only
a limited number of tuples in the output relation instead of
getting a long list (Example Query 1). At other times, one
might want to have the output relation sorted according to
the values of certain domains(Example Query 3). All these
desired results are incorporated in the system and the syntax
for the action statements is defined as follows-

    GET relation name.attribute name

    The action part of example Query 2 becomes -

    GET PART.P # , PART.PNAME

or   GET TOTAL(relation name SAME attribute names:attribute nam

or any combination of the above two. It is assumed that the
domain, if any, of which the totalling is required is the last
in the output list. The action part of example Query 5 becomes-

    GET PART.P# ,SUPPLY.J # ,TOTAL(SUPPLY SAME P# ,J# :QTY)

    For getting only a limited number (n) of tuples in the
output relation, the syntax is -

    GET (n) ....

    The action part for example Query 1 becomes -

    GET (3) PART.PNAME, PART.QOH

    For getting only the count of the number of tuples in
the work space, the syntax is -

    GET (COUNT) ....

    The action part for example Query 4 becomes -

    GET (COUNT) SUPPLY.P#

For getting the output in a sorted order, after all the output list, we add the information as follows -

GET .... ($\frac{UP}{DOWN}$ relation name.attribute name)

The action part for example Query 3 becomes -

GET PART.PNAME, PART.QOH (UP PART.QOH)

Here the first field in the UP/DOWN list is the major field, the next is the minor field and so on.

3.5 Complete Query:

After having defined the condition part and the action part of a query, the syntax of the complete query is as follows

Query name (Work space name)

CONDITION : condition part

ACTION : action part;

As an illustration the example Query 7 would be written as -

QUERY (W1)

CONDITION : SUPPLIER.S # < > SUPPLY.S #

ACTION : GET SUPPLIER.S# , SUPPLIER.SNAME;

In case of a multi-step query, the above three lines are repeated again except that the semicolon (;), indicating the end of the query, comes only at the end of the last subquery.

The complete syntax diagrams for the language just defined are given in Figure 3.2.

FIG. 3.2: SYNTAX DIAGRAMS OF LANGUAGE
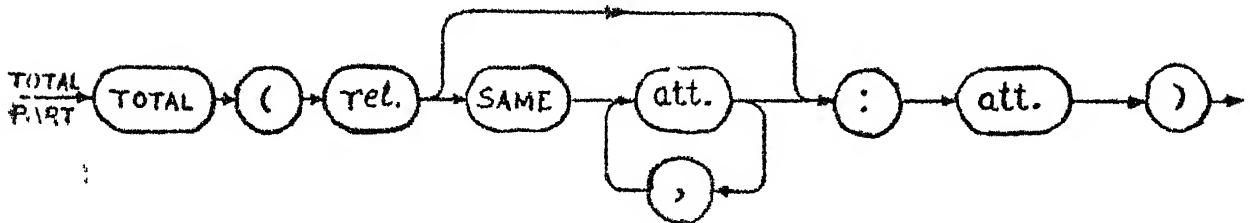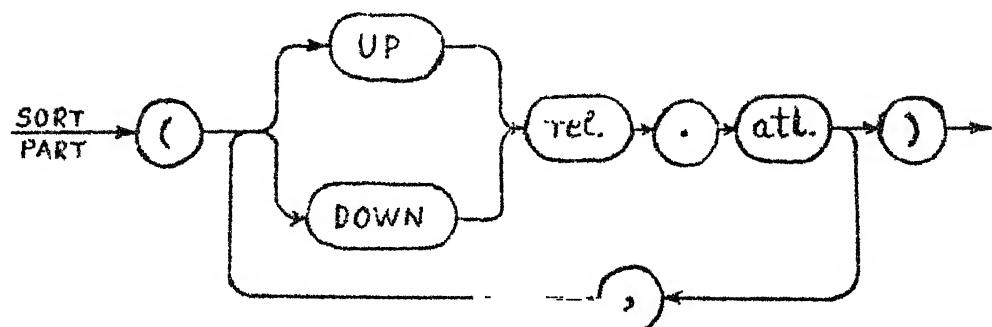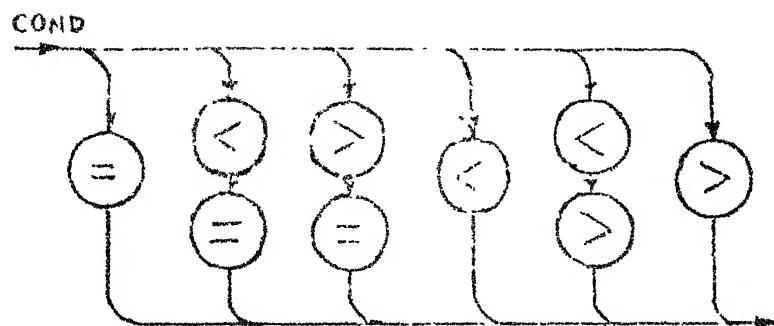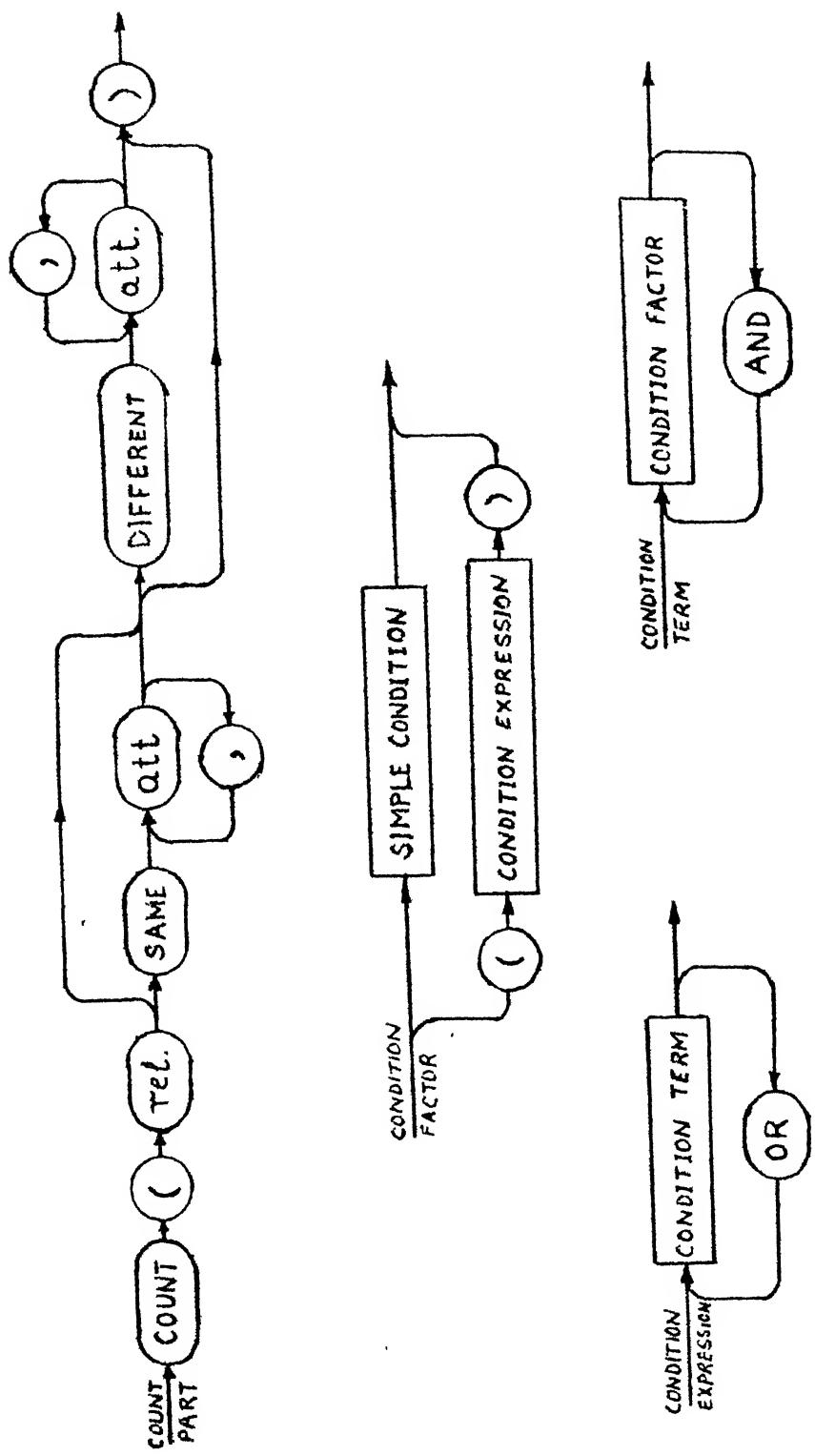
FIG. 3.2: SYNTAX DIAGRAMS OF LANGUAGE.

(Contd....)

FIG. 3.2: SYNTAX DIAGRAMS OF LANGUAGE.

(Contd...)

ACTION PART

GET

pos integer

COUNT

rel

att.

TOTAL PART

SORT PART

BLOCK

CONDITION

CONDITION EXPRESSION

ACTION

ACTION PART

QUERY

QUERY NAME

WORKSPACE NAME

BLOCK

QUERY PROGRAM

QUERY
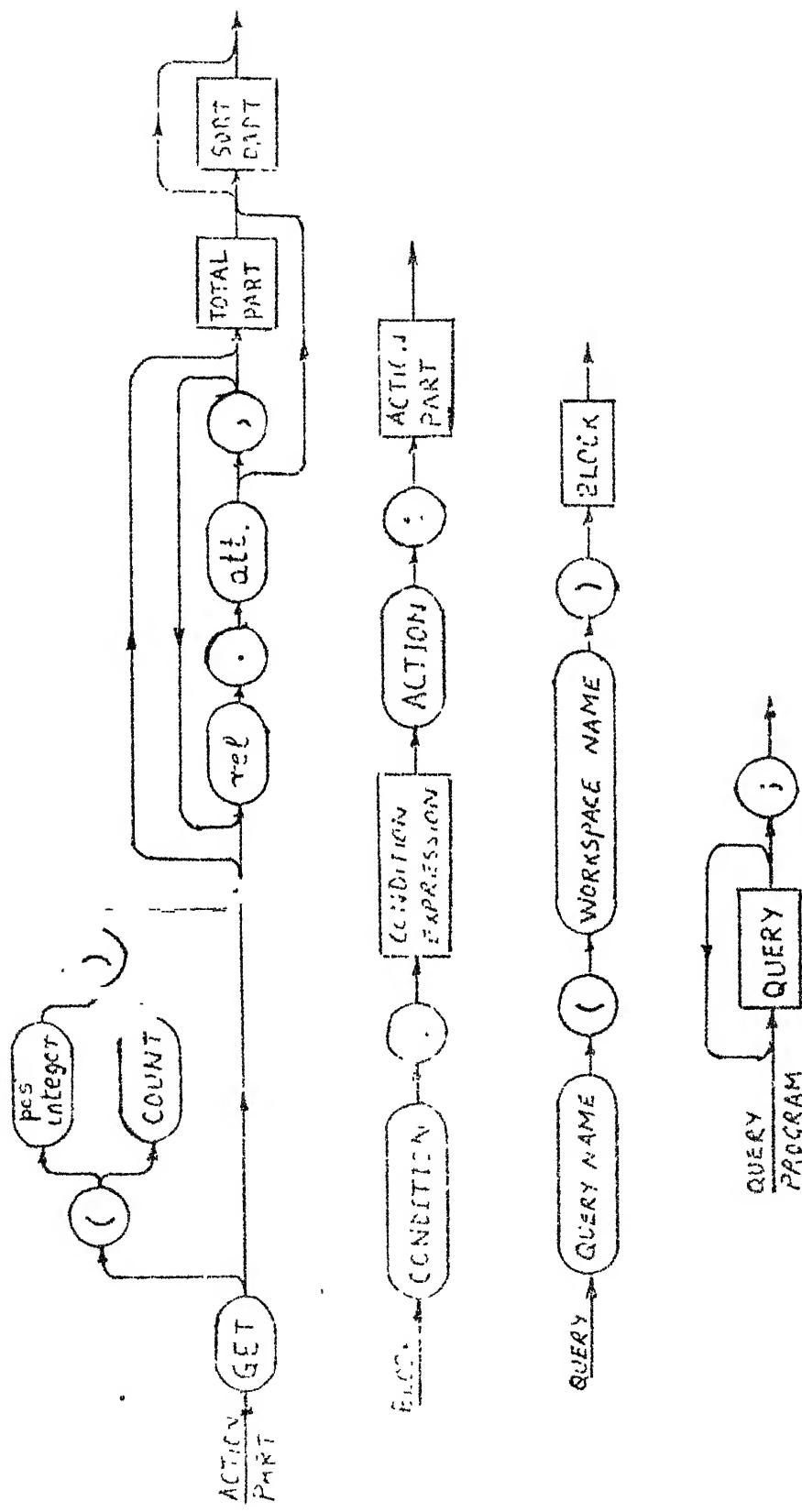
FIG. 3.2: SYNTAX DIAGRAMS OF LANGUAGE.

3.6 Completeness of the Language

To show that our language is reasonably complete, we will compare its features with the information retrieval features of DSL ALPHA. For each of these features of DSL ALPHA (reference [1]), we show that there exists a corresponding feature in our language which performs the same task as the one in DSL ALPHA.

(1)   The concepts of 'operation name' (GET) and 'workspace'(W) have been borrowed directly from DSL ALPHA. The user can name his own workspace and the 'name inheritance rules' of his workspace, as applicable in DSL ALPHA, are valid in our language also (Refer to example Query 10 in Chapter IV).

(2)   Corresponding to the 'target list' of DSL ALPHA we have the 'action part' in our language. The target list in DSL ALPHA contains the names of the attributes (qualified by the appropriate relation name) and so does the action part in our language. The target list can contain just a relation name, with no attributes specified, in case the whole of the relation has to be brought to the workspace. This feature is not present in our language. This does not limit the scope of our language, however, because the same purpose can be achieved by giving the complete list of attributes in that relation. (The information about the different attributes in each of the relations can be obtained from the 'RELDEF' file). Some functions which are allowed in the target list of DSL ALPHA are also

allowed in our language. Examples are 'COUNT' and 'TOTAL'.
The syntax for the use of these has already been described.
The functions 'MAX' and 'MIN' have not been included in the
action part but the purpose can be achieved by making a multi-
step query in our language. For example consider the follow-
ing query in DSL ALPHA -

> RANGE Z SUPPLY
>
> GET W MAX(Z.QTY) : (Z.S # =2)

This can be converted to the following in our language -

> QUERY(W1)
>
> CONDITION: SUPPLY.S # = " 2 "
>
> ACTION: GET SUPPLY.QTY
>
> QUERY(W)
>
> CONDITION: W1.QTY = 1 TOP
>
> ACTION: GET W1.QTY;

(3)   Corresponding to the 'qualification' in DSL ALPHA we
have the 'condition part' in our language. There is a major
difference, however, that the condition part is much less
procedural   than the qualification. A simple condition
in the condition part is entirely independent of what has to
come in the action part. Whereas the qualification (being
a qualifier of the target list) depends upon the target list.
As an example consider the following query -

> Find the supplier names of those suppliers who supply
> the part with part number 3.

The qualification part of the DSL ALPHA statement for this query becomes

RANGE SUPPLIER S

RANGE SUPPLY Z

GET W S.SNAME : (S.S# =Z.S#) A (Z.P# = 3)

In our language the condition part will be -

QUERY (W)

CONDITION: SUPPLY.S# = "3"

ACTION: GET SUPPLIER.SNAME;

(4)   The element ordering feature in DSL ALPHA is also included in our language. This expression follows the qualification in DSL ALPHA whereas it follows the action part of our language. This element ordering expression could be as general as in DSL ALPHA. (Refer to table ACTTAB described in Chapter IV).

(5)   The facility of specifying the 'quota' in DSL ALPHA is also included in our language. This is done by specifying the number of tuples to be retrieved within brackets after the reserved word GET in the action part of our language.(Refc to the table ACTTAB described in Chapter IV).

(6)   The negation sign ($\neg$) used in the qualification in DSL ALPHA, though not allowed in our language, yet the purpose can be achieved by using 'not equal to' sign (<>) and the boolean operators 'AND' and 'OR' in the condition part. As an example consider the following query -

Find the supplier numbers of suppliers who do not supply part number 3 (and may not supply any part at all also). The DSL ALPHA query for this becomes –

RANGE SUPPLIER S

RANGE SUPPLY Z ALL

GET W S.S : ¬ ((S.S# =Z.S#) ∧ (Z.P# =3))

The query in our language for this becomes –

QUERY (W)

CONDITION: SUPPLY.S# <> "3" OR SUPPLIER.S#<> SUPPLY.S#

ACTION: GET SUPPLIER.S# ;

(7)   Functions allowed in the qualification in DSL ALPHA are also allowed in our language. These functions are 'TOP' and 'BOTTOM'. The syntax for the use of these in our language has already been described.

(8)   The 'RANGE' statements in DSL ALPHA, provided for the purpose of abbreviation of relation names, are not allowed in out language. They just serve the purpose of avoiding the repetition of long relation names in the query statement, and are not functional. In as far as they also serve the purpose of defining the user's data sub-model, we can provide in our language the 'ENVIRONMENT' statement (Refer to Chapter VI). In DSL ALPHA the range statements also serve the purpose of allowing the shift of the universal quantifier from the qualification to these statements. This feature has been incorporated in our language by a special condition type (CNDCD = 3 (refer Chapter IV) which specifies that all the attribute

values of an attribute in a relation be treated as one entity.
For example consider the following query -

    Get the supplier numbers of suppliers who supply to all
    jobs.

The DSL ALPHA query for this becomes -

    RANGE SUPPLY Z

    RANGE JOB J ALL

    GET W Z.S # : (Z.J # = J.J #)

The corresponding query in our language becomes -

    QUERY (W)

    CONDITION : JOB J # = ALL

    ACTION : GET SUPPLY.S # ;

Thus we can say that our language incorporates all the
features of DSL ALPHA as described in reference [1].

CHAPTER IV

SOURCE QUERY TO OBJECT QUERY

4.1 Source Query and Object Query:

After having designed the query language, we have to
build a system that accepts queries in this language and
executes them. We call, for notation sake, a query in our
new language as the 'Source Query' and that converted to
one having relational algebra operations as 'Object Query'.
Like any other language translator, the first phase of this
system analyses the string of characters in the source query
so as to form meaningful symbols. This phase is called
'lexical analysis'. Outcome of this phase is a string of
meaningful primitives, called lexemes (like 'GET', ';', 'ALL'
etc. in our language). The next phase of the system has to
combine such lexemes, collected from the input string of
characters, and recognize grammatical phrases in the source
query. This is called 'syntax analysis'. Finally, the system
has to discover how the source query does what it is supposed
to do. This is the 'semantic analysis' phase. It is required
to build certain tables which store information contained
in the source query (about the intent and the names of relation
and attributes involved). From the information stored in these
tables and that already existing in the system (about various

relations and their attributes present in the data base),
the object query is constructed. The object query has
statements like

> JOIN   Rl AND R2 OR Al;
>
> or   PROJECT Rl BY Al   A2:
>
> or   SORT Rl ON ASCENDING Al DESCENDING A2; etc.

Here Rl, R2 are relation names, either original or those
constructed by the user for storing intermediate results,
and Al, A2 are attribute names.

4.2 Lexical Analysis:

The task of the lexical analyser is to recognize in
the input string, which is the source query, items like
names, numbers and separators. This task is done by pro-
cedure GETNEXTSYM. Whenever the syntax analyser makes a
call to this procedure, the next symbol (lexeme) in the
input string is passed on to the calling procedure through
a global variable 'SYM'. 'GETCH' is a standard procedure
which reads the input text and passes on to the calling
procedure (GETNEXTSYM in this case) the next character in
the string. The global variable for a character is 'CH'.

NORW, the number of reserved words in the language,
is a constant and is set to 14 in the beginning of the
program. These reserved words are ACTION, ALL, AND, BOTTOM,
CONDITION, COUNT, DIFFERENCE, DOWN, GET, OR, SAME, TOP, TOTA

and UP. There are 11 delimiters. They are -':'(colon),
'(' (lparen), ')'(rparen), '.'(period), ','(comma), ';'(semi-
colon), '='(eql), '<'(lss), '>'(gtr), '"'(aspst) and '*'(star)
Three other delimiters are formed by a combination of these.
They are - '<>' (neq), '<='(leq) and '>= (geq). The reserved
words and delimiters are initialized by procedure INITIALIZE
(Fig. 4.1).

The identifiers in the language could be upto 'AL'
characters in length. (AL is a constant and is set to 10 in the
program). As usual, the identifiers can start with any of the
alphabets ('A' to 'Z') and can contain alphabets, digits and
characters '#' and '-'. These last two characters have been
included because attribute names in a data base can contain
such characters. (e.g. 'MGR-E#').

Numbers could be integers or floating point reals. A
negative number can be represented by a '-' sign before the
value, but a non-negative number should not have a '+' sign
before it.

The listing of procedures GETNEXTSYM and GETCH are
given in Figure 4.2.

4.3 Syntax Analysis:

The syntax of the source query is checked in accordance
with the syntax diagrams of the language described in Chapter III
It is almost a 'one look-ahead' parsing scheme. Provision has

```
rocedure INITIALIZE;
  begin WORD[ 1]:='ACTION      ';
    WORD[ 2]:='ALL         ';
    WORD[ 3]:='AND         ';
    WORD[ 4]:='BOTTOM      ';
    WORD[ 5]:='CONDITION   ';
    WORD[ 6]:='COUNT       ';
    WORD[ 7]:='DIFFERENT   ';
    WORD[ 8]:='DOWN        ';
    WORD[ 9]:='GET         ';
    WORD[10]:='OR          ';
    WORD[11]:='SAME        ';
    WORD[12]:='TOP         ';
    WORD[13]:='TOTAL       ';
    WORD[14]:='UP          ';
    WSYM[ 1]:=ACTIONSYM      ;
    WSYM[ 2]:=ALLSYM         ;
    WSYM[ 3]:=ANDSYM         ;
    WSYM[ 4]:=BOTTOMSYM      ;
    WSYM[ 5]:=CONDITIONSYM   ;
    WSYM[ 6]:=COUNTSYM       ;
    WSYM[ 7]:=DIFFERENTSYM   ;
    WSYM[ 8]:=DOWNSYM        ;
    WSYM[ 9]:=GETSYM         ;
    WSYM[10]:=ORSYM          ;
    WSYM[11]:=SAMESYM        ;
    WSYM[12]:=TOPSYM         ;
    WSYM[13]:=TOTALSYM       ;
    WSYM[14]:=UPSYM          ;
    for CH:=CHR(0) to CHR(127) do SSYM[CH]:=NUL;
    SSYM[':']:=COLON      ;
    SSYM['(']:=LPAREN     ;
    SSYM[')']:=RPAREN     ;
    SSYM['.']:=PERIOD     ;
    SSYM[',']:=COMMA      ;
    SSYM[';']:=SEMICOLON  ;
    SSYM['=']:=EQL        ;
    SSYM['<']:=LSS        ;
    SSYM['>']:=GTR        ;
    SSYM['"']:=APST       ;
    SSYM['*']:=STAR       ;
    CC:=0;
    LL:=0;
    CH:=' ';
    KK:=AL;
    ERR:=0;
    TEMPSYM:=NUL;
    IREL:=NORELS
  end;
```

```
procedure GETNEXTSYM;
  var
    I,J,K:integer;
  procedure GETCH;
    begin
      if CC=LL then
        begin
          if EOF(INPUT) then
            begin WRITELN('Query is incomplete.');
              HALT
            end;
          LL:=0;
          CC:=0;
          WRITE(' ');
          while not EOLN(INPUT) do
            begin LL:=LL+1;
              READ(CH);
              WRITE(CH);
              LINE[LL]:=CH
            end;
          WRITELN;
          LL:=LL+1;
          READ(LINE[LL])
        end;
      CC:=CC+1;
      CH:=LINE[CC]
    end;
  begin  while CH=' ' do GETCH;
    if CH in ['A'..'Z'] then
      begin K:=0;
        repeat
          if K<AL then
            begin K:=K+1;
              A[K]:=CH
            end;
          GETCH
        until not (CH in ['A'..'Z','0'..'9','#','-']);
        if K>=KK then KK:=K
        else
        repeat A[KK]:=' ';
          KK:=KK-1
        until KK=K;
        ID:=A;
        I:=1;
        J:=NORW;
        repeat
          K:=(I+J) div 2;
          if ID<=WORD[K] then J:=K-1;
          if ID>=WORD[K] then I:=K+1
        until I>J;
        if I-1>J then SYM:=NSYM[K]
        else SYM:=IDENT
      end
```

FIGURE 4.9 PROCEDURE GETNEXTSYM   (CONTD.)

```
else
  if (CH in ['0'..'9']) or (CH='-') then
    begin INTNUM:=0;
      REALNUM:=0.0;
      SYM:=INTNUMBER;
      if CH='-' then
        begin TEMPCH:='-';
          GETCH
        end;
      if CH in ['0'..'9'] then
        begin
          repeat INTNUM:=10*INTNUM+(ORD(CH)-ORD('0'));
            GETCH
          until not (CH in ['0'..'9']);
          if CH='.' then
            begin REALNUM:=INTNUM;
              INTNUM:=0;
              GETCH;
              SYM:=REALNUMBER;
              P:=10;
              while CH in ['0'..'9'] do
                begin REALNUM:=REALNUM+(ORD(CH)-ORD('0'))/P;
                  P:=P*10;
                  GETCH
                end
            end
        end
      else ERROR(17);
      if TEMPCH='-' then
        begin REALNUM:=-REALNUM;
          INTNUM:=-INTNUM;
          TEMPCH:=' '
        end
    end
  else
    if CH='<' then
      begin GETCH;
        if CH='>' then
          begin SYM:=NEQ;
            GETCH
          end
        else
          if CH='=' then
            begin SYM:=LEQ;
              GETCH
            end
          else SYM:=LSS
      end
    else
      if CH='>' then
        begin GETCH;
          if CH='=' then
            begin SYM:=GEQ;
              GETCH
            end
          else SYM:=GTR
        end
      else
        begin SYM:=SSYM[CH];
          GETCH
        end
end;
```

FIGURE 4.2 PROCEDURE GETNEXTSYM

been made to indicate to the user precise error messages in case there are errors in the structure of the query. Two procedures, ERROR and WARN have been written for the purpose. A listing of these is given in Figure 4.3. In case of errors the source query is checked as far as possible for any further errors and the user is indicated of the same. The system does not produce the object query in case of syntactical errors in source query.

A procedure named CHECKQUERY is called by the main program to check the syntax of the complete source query. It also builds up a number of tables required for producing the object query at a later stage. This procedure in turn calls two procedures CONDEXP (to process the condition part) and ACT (to process the action part). CONDFAC is a procedure used by CONDEXP to check the syntax of each simple condition in the condition part. Listings of CONDEXP and ACT are given in Figures 4.4 and 4.5 respectively. Listing of CHECKQUERY is given in Figure 4.6. Now we describe the various tables the system has and builds during the process of syntax analysis.

4.4 Tables Maintained (Directories):

For a given data base information about the different relations and attributes of the system is kept stored in the form of tables of records. These tables are referred to very frequently during the process of running a query. We will describe these in detail.

```pascal
procedure ERROR(N:integer);
  begin WRITELN('*',' ':CC-1,'^',N:2);
    case N of
      1: WRITELN('** Illegal Attribute name.');
      2: WRITELN('** Non negative integer expected.');
      3: WRITELN('** Comparision operator expected.');
      4: WRITELN('** Unpaired paranthesis.');
      5: WRITELN('** "<>" expected.');
      6: WRITELN('** Illegal relation name.');
      7: WRITELN('** Left paranthesis expected.');
      8: WRITELN('** Period expected.');
      9: WRITELN('** "TOP" or "BOTTOM" expected.');
     10: WRITELN('** Illegal symbol.');
     11: WRITELN('** "GET" expected.');
     12: WRITELN('** Colon expected.');
     13: WRITELN('** "UP" or "DOWN" expected.');
     14: WRITELN('** "CONDITION" expected.');
     15: WRITELN('** Illegal work space name.');
     16: WRITELN('** Illegal attribute value.');
     17: WRITELN('** Number (real or integer) expected.');
     18: WRITELN('** "=" expected.');
     19: WRITELN('** " " " (APOSTROPHE) expected.');
     20: WRITELN('** "ACTION" expected.')
    end;
    ERR:=ERR+1
  end;

procedure WARN(N:integer);
  begin WRITELN;
    WRITELN('Warning no.',N:2);
    case N of
      1: WRITELN('-- Missing semicolon at end of query.');
      2: WRITELN('-- Totalling/Sorting not done in output relation
    end;
    WRITELN
  end;

procedure GETNAME(var ID:ALFA);
  var
    I,J,K:integer;
    CH:char;
    begin READ(F,CH);
    while CH=' ' do READ(F,CH);
    I:=1;
    while ((CH#' ')and(I<=AL)) do
      begin ID[I]:=CH;
        I:=I+1;
        READ(F,CH)
      end;
    for J:=I to AL do ID[J]:=' '
  end;
```

FIGURE 4.3 PROCEDURES ERROR, WARN
AND GETNAME

```
procedure CONDEXP;
    procedure ADDGAMA;
        begin NEW(GP2);
          GP2^.NEXTGAMA:=GP1;
          GP1:=GP2
        end;
    procedure CONDTERM;
        procedure CONDFAC;
          begin
          if SYM in [COUNTSYM,TOTALSYM,IDENT] then
            begin CN:=CN+1;
            ADDGAMA;
            GP1^.GAMAVAL:=CN;
            GP1^.GAMASYM:=INTNUMBER;
            with CONTAB[CN] do
              begin
              if SYM=COUNTSYM then
                begin CNDCD:=9;
                ATTNOC:=0;
                GETNEXTSYM;
                if SYM=LPAREN then
                  begin GETNEXTSYM;
                  if SYM=IDENT then
                    begin GETRELNO(RELNO,ID);
                    if RELNO>0 then
                      begin RELNOC:=RELNO;
                      GETNEXTSYM;
                      if SYM=SAMESYM then
                        begin NEW(P1);
                        SATTS9:=P1;
                        repeat GETNEXTSYM;
                          if SYM=IDENT then
                            begin GETATTNO(ATTNO,ID);
                            if ATTNO>0 then
                              begin P11:=P1;
                              NEW(P1);
                              P11^.NEXTATT:=P1;
                              P11^.ATTNOL:=ATTNO;
                              GETNEXTSYM
                            end
                            else ERROR(1)
                          end
                          else ERROR(1)
                        until SYM=COMMA;
                        P11^.NEXTATT:=nil
                      end;
                    if SYM=DIFFERENTSYM then
                      begin NEW(P1);
                      DATTS9:=P1;
                      repeat GETNEXTSYM;
                        if SYM=IDENT then
                          begin GETATTNO(ATTNO,ID);
                          if ATTNO>0 then
                            begin P11:=P1;
                            NEW(P1);
                            P11^.NEXTATT:=P1;
                            P11^.ATTNOL:=ATTNO;
                            GETNEXTSYM
                          end
                          else ERROR(1)
                        end
                        else ERROR(1)
                      until SYM=COMMA;
                      P11^.NEXTATT:=nil
                    end;
```

FIGURE 4.4 PROCEDURE CONDEXP   (CONTD.

```
            if SYM=RPAREN then
             begin GETNEXTSYM;
              if SYM in [EQL,LSS,GTR,NEQ,LEQ,GEQ] then
               begin COND:=SYM;
                GETNEXTSYM;
                 if SYM=INTNUMBER then
                  begin VAL9:=INTNUM;
                   if VAL9<0 then ERROR(2);
                   GETNEXTSYM
                  end
                 else ERROR(2)
                end
               else ERROR(3)
              end
             else ERROR(4)
            end
           else ERROR(6)
          end
         else ERROR(6)
         end
       else ERROR(7)
      end
     else
      if SYM=TOTALSYM then
       begin CNDCD:=8;
        ATTNOC:=0;
        GETNEXTSYM;
        if SYM=LPAREN then
         begin GETNEXTSYM;
          if SYM=IDENT then
           begin GETRELNO(RELNO,ID);
            if RELNO>0 then
             begin RELNOC:=RELNO;
              GETNEXTSYM;
              if SYM=SAMESYM then
               begin NEW(P1);
                SATTS8:=P1;
                repeat GETNEXTSYM;
                 if SYM=IDENT then
                  begin GETATTNO(ATTNO,ID);
                   if ATTNO>0 then
                    begin P11:=P1;
                     NEW(P1);
                     P11^.NEXTATT:=P1;
                     P11^.ATTNOL:=ATTNO;
                     GETNEXTSYM
                    end
                   else ERROR(1)
                  end
                 else ERROR(1)
                until SYM#COMMA;
                P11^.NEXTATT:=nil
               end;
```

FIGURE 6.4 PROCEDURE CONDEXP   (CONTD.

```
                    if SYM=COLON then
                     begin GETNEXTSYM;
                      if SYM=IDENT then
                       begin GETATTNO(ATTNO,ID);
                        if ATTNO>0 then
                         begin TATT8:=ATTNO;
                          GETNEXTSYM;
                          if SYM=RPAREN then
                           begin GETNEXTSYM;
                            if SYM in [EQL,LSS,GTR,NEQ,LEQ,GEQ]
                             begin COND:=SYM;
                              GETNEXTSYM;
                              if SYM=INTNUMBER then
                               begin TOID8:=INTNUMBER;
                                VAL8I:=INTNUM;
                                GETNEXTSYM
                               end
                              else
                               if SYM=REALNUMBER then
                                begin TOID8:=REALNUMBER;
                                 VAL8R:=REALNUM;
                                 GETNEXTSYM
                                end
                               else ERROR(17)
                             end
                            else ERROR(3)
                           end
                          else ERROR(4)
                         end
                        else ERROR(1)
                       end
                      else ERROR(1)
                     end
                    else ERROR(12)
                   end
                  else ERROR(6)
                 end
                else ERROR(6)
               end
              else ERROR(7)
             end
            else
             if SYM=IDENT then
              begin GETRELNO(RELNO,ID);
               if RELNO>0 then
                begin RELNOC:=RELNO;
                 GETNEXTSYM;
                 if SYM=PERIOD then
                  begin GETNEXTSYM;
                   if SYM=IDENT then
                    begin GETATTNO(ATTNO,ID);
                     if ATTNO>0 then
                      begin ATTNOC:=ATTNO;
                       GETNEXTSYM;
                       if SYM in [EQL,NEQ,LSS,LEQ,GTR,GEQ]
                        begin COND:=SYM;
                         GETNEXTSYM;
                         if SYM=ALLSYM then
                          begin CNDCD:=3;
                           if COND#EQL then ERROR(18);
                           GETNEXTSYM
                          end
```

FIGURE 4.4 PROCEDURE CONDEXP   (CONTD.

```
        else
          if SYM=STAR then
          begin CNDCD:=2;
            if COND#EQL then ERROR(18);
            GETNEXTSYM
          end
          else
          if SYM=IDENT then
          begin GETRELNO(RELNO,ID);
            if RELNO>0 then
            begin CNDCD:=7;
              RELNO7:=RELNO;
              GETNEXTSYM;
              if SYM=PERIOD then
              begin GETNEXTSYM;
                if SYM=IDENT then
                begin GETATTNO(ATTNO,ID);
                  if ATTNO>0 then
                  begin ATTNO7:=ATTNO;
                    if COND#NEQ then ERROR(5);
                    if ATTNO7#ATTNOC then
                      ERROR(1);
                    GETNEXTSYM
                  end
                  else ERROR(1)
                end
                else ERROR(1)
              end
              else ERROR(8)
            end
            else ERROR(6)
          end
          else
          if SYM=INTNUMBER then
          begin VAL4:=INTNUM;
            GETNEXTSYM;
            if SYM=TOPSYM then
            begin CNDCD:=4;
              GETNEXTSYM
            end
            else
            if SYM=BOTTOMSYM then
            begin CNDCD:=5;
              GETNEXTSYM
            end
            else ERROR(9)
          end
          else
          if SYM=APST then
          begin CNDCD:=1;
            GETNEXTSYM;
            if SYM in
            [IDENT,INTNUMBER,REALNUMBER]then
            begin TOID1:=SYM;
              case TOID1 of
                IDENT:VAL1I:=SEARCH(ATTNOC,ID)
                REALNUMBER:VAL1R:=REALNUM;
                INTNUMBER:VAL1I:=INTNUM
              end;
```

FIGURE 4.4 PROCEDURE CONDEXP   (CONTD.)

```
                                        GETNEXTSYM;
                                        if SYM=APST then
                                          GETNEXTSYM
                                        else ERROR(19)
                                      end
                                    else ERROR(16)
                                  end
                                else ERROR(10)

                    end
                  else ERROR(3)
                end
              else ERROR(1)
            end
          else ERROR(1)
        end
      else ERROR(8)
    end
  else ERROR(6)
end

        end
      end
      else
        if SYM=LPAREN then
          begin ADDGAMA;
            GP1^.GAMASYM:=LPAREN;
            GETNEXTSYM;
            CONDEXP;
            if SYM=RPAREN then
              begin ADDGAMA;
                GP1^.GAMASYM:=RPAREN;
                GETNEXTSYM
              end
            else ERROR(4)
          end
        else ERROR(10)
  end;
  begin CONDFAC;
    while SYM=ANDSYM do
      begin ADDGAMA;
        GP1^.GAMASYM:=ANDSYM;
        GETNEXTSYM;
        CONDFAC
      end
  end;
begin CONDTERM;
  while SYM=ORSYM do
    begin ADDGAMA;
      GP1^.GAMASYM:=ORSYM;
      GETNEXTSYM;
      CONDTERM
    end
end;
```

FIGURE 4.4 PROCEDURE CONDEXP

```
procedure ACT;
  var
    I,J,K:integer;
    FLAG:boolean;
  procedure INSERT(var I:integer);
    var
      J:integer;
      FLAG:boolean;
    begin J:=1;
      FLAG:=false;
      repeat
        if TABATT[J].ATTNO=ATTNO then FLAG:=true;
        J:=J+1
      until (FLAG or (TABATT[J].ATTNO=0))
      with TABATT[I] do
        begin
          if FLAG then RELNO:=0
          else
            begin RELNO:=RELNO;
              ATTNO:=ATTNO;
              ACTCD:=0;
              I:=I+1
            end
        end
    end;
  begin I:=1;
    if SYM=GETSYM then
      begin GETNEXTSYM;
        if SYM=LPAREN then
          begin GETNEXTSYM;
            if SYM=COUNTSYM then
              begin ACTTAB.COUNTREQ:=true;
                GETNEXTSYM
              end
            else
              if SYM=INTNUMBER then
                begin ACTTAB.LIMNO:=INTNUM;
                  ACTTAB.LMTREQ:=true;
                  GETNEXTSYM
                end
              else ERROR(10);
            if SYM=RPAREN then GETNEXTSYM
            else ERROR(4)
          end;
```

FIGURE 4.5 PROCEDURE ACT

then

```
if SYM=IDENT then
  begin GETRELNO(RELNO,ID);
    if RELNO>0 then
      begin TEMPSYM:=IDENT;
        TABATT[I].RELNOT:=RELNO;
        TABATT[I].ACTCD:=1;
        GETNEXTSYM;
        if SYM=PERIOD then
          begin GETNEXTSYM;
            if SYM=IDENT then
              begin GETATTNO(ATTNO,ID);
                if ATTNO>0 then
                  begin TABATT[I].ATTNOT:=ATTNO;
                  J:=I+1;
                  GETNEXTSYM;
                  while SYM=COMMA do
                    begin GETNEXTSYM;
                      if SYM=IDENT then
                      begin GETRELNO(RELNO,ID);
                        if RELNO>0 then
                          begin TABATT[I].RELNOT:=RELNO;
                          GETNEXTSYM;
                          if SYM=PERIOD then
                            begin GETNEXTSYM;
                              if SYM=IDENT then
                              begin GETATTNO(ATTNO,ID);
                                if ATTNO>0 then
                                  begin TABATT[I].ATTNOT:=ATTNO;
                                  TABATT[I].ACTCD:=1;
                                  I:=I+1;
                                  GETNEXTSYM
                                  end
                                else ERROR(1)
                              end
                              else ERROR(1)
                            end
                            else ERROR(8)
                          end
                          else ERROR(6)
                        end
                      end
                    end
                  end
                else ERROR(1)
                end
              else ERROR(1)
            end
          else ERROR(8)
          end
        else ERROR(6)
        end;
```

FIGURE 4.5 PROCEDURE ACT (CONTD)

```
if SYM=TOTALSYM then
  begin GETNEXTSYM;
   ACTTAB.TTLREQ:=true;
   if SYM=LPAREN then
     begin GETNEXTSYM;
      if SYM=IDENT then
        begin GETRELNO(RELNO,ID);
         if RELNO>0 then
           begin ACTTAB.TREL:=RELNO;
            GETNEXTSYM;
            if SYM=SAMESYM then
              begin NEW(P1);
               ACTTAB.SATTS:=P1;
               repeat GETNEXTSYM;
                 if SYM=IDENT then
                   begin GETATTNO(ATTNO,ID);
                    if ATTNO>0 then
                      begin P11:=P1;
                       INSERT(I);
                       NEW(P1);
                       P11^.NEXTATT:=P1;
                       P11^.ATTNOL:=ATTNO;
                       GETNEXTSYM
                      end
                    else ERROR(1)
                   end
                 else ERROR(1)
               until SYM#COMMA;
               P11^.NEXTATT:=nil
              end;
            if SYM=COLON then
              begin GETNEXTSYM;
               if SYM=IDENT then
                 begin GETATTNO(ATTNO,ID);
                  if ATTNO>0 then
                    begin ACTTAB.TATT:=ATTNO;
                     TABATT[I].RELNOT:=RELNO;
                     TABATT[I].ATTNOT:=ATTNO;
                     TABATT[I].ACTCD:=1;
                     I:=I+1;
                     GETNEXTSYM;
                     if SYM=RPAREN then
                       GETNEXTSYM
                     else ERROR(4)
                    end
                  else ERROR(1)
                 end
               else ERROR(1)
              end
            else ERROR(12)
           end
         else ERROR(6)
        end
      else ERROR(6)
     end
   else ERROR(7)
  end
```

FIGURE 4.5 PROCEDURE ACT

```
else
  if TEMPSYM<>IDENT then ERROR(10);
  TEMPSYM:=NUL;
  if SYM=LPAREN then
   begin ACTTAB.SRTREQ:=true;
    NEW(SP1);
    ACTTAB.PSORT:=SP1;
    repeat GETNEXTSYM;
      if SYM in [UPSYM,DOWNSYM] then
       begin
        if SYM=UPSYM then TEMPSORT:=UP
        else TEMPSORT:=DOWN;
        GETNEXTSYM;
        if SYM=IDENT then
         begin GETRELNO(RELNO,ID);
          if RELNO>0 then
           begin SP11:=SP1;
            NEW(SP1);
            SP11^.SORDER:=TEMPSORT;
            SP11^.NEXTSORT:=SP1;
            GETNEXTSYM;
            if SYM=PERIOD then
             begin GETNEXTSYM;
              if SYM=IDENT then
               begin GETATTNO(ATTNO,ID);
                if ATTNO>0 then
                 begin SP11^.ATTNOS:=ATTNO;
                  INSERT(I);
                  GETNEXTSYM
                 end
                else ERROR(1)
               end
              else ERROR(1)
             end
            else ERROR(8)
           end
          else ERROR(6)
         end
        else ERROR(6)
       end
      else ERROR(13)
    until SYM#COMMA;
    SP11^.NEXTSORT:=nil;
    if SYM=RPAREN then GETNEXTSYM
    else ERROR(4)
   end
 end
else ERROR(11)
end;
```

FIGURE 4.5 PROCEDURE ACT

```
ocedure CHECKQUERY;
  procedure BLOCK;
    procedure COMPEXP;
    PROCEDURE ACT;
    begin
      if SYM=CONDITIONSYM then
        begin GETNEXTSYM;
          if SYM=COLON then
            begin GETNEXTSYM;
              CN:=0;
              GP1:=nil;
              COMPEXP
            end
          else ERROR(12)
        end
      else ERROR(14);
      if SYM#ACTIONSYM then
        begin while SYM#ACTIONSYM do GETNEXTSYM;
          ERROR(20)
        end;
      GETNEXTSYM;
      if SYM=COLON then
        begin GETNEXTSYM;
          ACT
        end
      else ERROR(12)
    end;
  begin
    if SYM=IDENT then GETNEXTSYM;
    if SYM=LPAREN then
      begin GETNEXTSYM;
        if SYM=IDENT then
          begin WSPACE:=ID;
            GETNEXTSYM;
            if SYM=RPAREN then
              begin GETNEXTSYM;
                BLOCK
              end
            else ERROR(4)
          end
        else ERROR(15)
      end
    else ERROR(7)
  end;
```

FIGURE 4.6 PROCEDURE CHECKQUERY

4.4.1 <u>Directory of Attributes in the System</u>

This is named ATTDIR. It is an array of dimension
NOATTS, the number of attributes in the system. Each element
of the array is a record of the type ATTRIBUTE. This record
has two fields —

ATTNAMED : for the name of the attribute;

and   TYPEATT : for the type of the field, i.e., alphanumeric
              (IDENT), integer (INTNUMBER) or real
              (REALNUMBER).

NOATTS, the number of attributes in the data base,
is declared as a constant in the main program. For our example
system it is set equal to 12. It has to be changed for a
different data base. The specifications of the attributes are
read in from a file named 'ATTDEF' by a procedure DEFINEATTS.
A listing of this procedure is given in Fig. 4.7. Each line
of the file ATTDEF should define one attribute. The details
should be given in the following order — attribute name and
the type of the attribute ('I' for integer, 'R' for real
and 'A' for alphanumeric). There should be NOATTS attri-
butes in the file. The order in which these are given is
immaterial. The system stores these attributes in ATTDIR in
the same order. Whenever an attribute name is encountered
in the source query, a procedure GETATTNO gets the sequence
number of the attribute (ATTNO) from the table ATTDIR and
uses this number for storing in the runtime tables. A listing

```pascal
procedure DEFINEATTS;
    var
      I:integer;
      TP:char;
    begin
      RESET(F,'ATTDEF');
      for I:=1 to NOATTS do
        with ATTDIR[I] do
          begin GETNAME(ID);
            ATTNAMED:=ID;
            READ(F,TP,NOCHARS);
            if TP in ['I','R','A'] then
            case TP of
              'I':TYPEATT:=INTNUMBER;
              'R':TYPEATT:=REALNUMBER;
              'A':TYPEATT:=IDENT
            end
            else
              begin WRITELN('** Error in ATTDEF -',ATTNAMED:AL);
                HALT
              end
          end
    end;

  procedure GETATTNO(var N:integer;ID:ALFA);
    var
      I:integer;
    begin I:=1;
      while((I<=NOATTS-1)and(ATTDIR[I].ATTNAMED<>ID)) do I:=I+1;
      if((I<NOATTS)or((I=NOATTS)and(ATTDIR[I].ATTNAMED=ID))) then N:=I
      else N:=0
    end;

  procedure READATTS;
    var
      I,J,K:integer;
      ID:ALFA;
    begin RESET(F,'TABLES');
      for I:=1 to NOATTS do
        begin
          if ATTDIR[I].TYPEATT#INTNUMBER then
            begin GETNAME(ID);
              if ID#ATTDIR[I].ATTNAMED then
                begin ERROR(2);
                  HALT
                end
              else
                begin READ(F,J);
                  for K:=1 to J do
                    begin GETNAME(ID);
                      TABLE[I][K]:=ID
                    end
                end
            end
        end
    end;

  function SEARCH(N:integer;ID:ALFA):integer;
    var
      I:integer;
      FLAG:boolean;
    begin I:=1;
      FLAG:=false;
      while not(FLAG or (I>AL)) do
        begin
          if ID=TABLE[N][I] then FLAG:=true
          else I:=I+1
        end;
      if FLAG then SEARCH:=I
      else ERROR(16)
    end;
```

FIGURE 4.7. PROCEDURES DEFINEATTS, GETATTNO,
READATTS AND FUNCTION SEARCH

of procedure GETATTNO is given in Figure 4.7. A number zero is returned by the procedure in case the identifier in the source query, which is supposed to be an attribute name (as per the syntax of the language), does not match with any of the attribute names stored in the directory. An error message is printed out in the case.

For each attribute which is not of the type integer, we read in a table which associates a unique integer number with each different type of value for that attribute. These tables are read in from a file 'TABLES' by a procedure READATTS. A listing of this procedure is given in Fig.4.7. The file 'TABLES' should contain the name of the attribute, the number of different values of the attribute and the actual values of the attributes for each of the attributes which is not of the type integer.

4.4.2 Directory of Relations Stored in the System:

This is named RELDIR. It is an array of dimension NORELSP. NORELS is the number of relations originally stored in the system.

$$NORELSP = NORELS + 20.$$

This number is chosen such that we can store all the intermediate relations built by the user. Each element of the array RELDIR is a record of the type RELATION. This record has six fields –

RELNAMED : for the name of the relation;

ATTD : for the attributes in the relation. This is an array of integers (one integer for each attribute) of dimension NOATTS (to take care of the extreme case when there is a relation in the system which contains all the attributes in the system). These integers are the same as the sequence numbers of the attributes in ATTDIR. Order is maintained and a 'ZERO' means no more attributes beyond that index;

KEYD : for the key fields of the relation. This is also an array of the same type as ATTD;

FRSTPL : is a pointer to the first tuple in the relation. (Each tuple is a record of NOATTS fields. In addition it has a MARKing facility, used at the time of implementing the algebraic operations, and a pointer to the next tuple in the relation.);

NTPLS : for storing the number of tuples in the relation;

and RELFL : for storing the name of the file in which the relation is stored.

NORELS, the number of relations defined in the data base and NORELSP, the total number of relations permitted in the

system are also declared as constants in the main program. For
our example system they are set equal to 4 and 24 respectively.
These have to be changed for a different data base. The speci-
fications of the relations are read in from a file 'RELDEF'
by a procedure DEFINERELS. A listing of this procedure is
given in Fig. 4.8. Each relation should be defined in this
file. The details should be given in the following order -
relation name, the number of attributes in the relation, the
names of the attributes in the relation, the number of fields
which together form the key to the tuples in the relation,
the attribute names which form the key, the number of tuples
stored in the relation and finally the name of the file which
contains the relation. There should be definitions of NORELS
relations in this file. As in the case of ATTDIR, the order
of relations in this file is immaterial. The system stores
these relations in the same order in RELDIR. Whenever a relation
name is encountered in the source query, a procedure GETRELNO
gets the sequence number of the relation (RELNO) from RELDIR
and uses this number for storing in the runtime tables. A
listing of procedure GETRELNO is given in Fig. 4.8. An illegal
relation name in the source query causes an error message to
be printed out for the user.

Each of the relations in the system is stored in a file
each. The names of these files is read from the file 'RELDEF'
and stored in directory RELDIR as described above. The relations

```
procedure DEFINERELS;
  var
    I,J,K:integer;
    ID:ALFA;
  begin RESET(F,'RELDEF');
    for I:=1 to NORELS do
      with RELDIR[I] do
        begin GETNAME(ID);
          RELNAMED:=ID;
          READ(F,J);
          for K:=1 to J do
            begin GETNAMED(ID);
              GETATTNO(ATTNO,ID);
              if ATTNO=0 then
                begin WRITELN('** Error in RELDEF -',RELNAMED:AL);
                  HALT
                end;
              ATTD[K]:=ATTNO
            end;
          for K:=J+1 to NOATTS do ATTD[K]:=0;
          READ(F,J);
          for K:=1 to J do
            begin GETNAMED(ID);
              GETATTNO(ATTNO,ID);
              if ATTNO=0 then
                begin WRITELN('** Error in RELDEF -',RELNAMED:AL);
                  HALT
                end;
              KEYD[K]:=ATTNO
            end;
          for K:=J+1 to NOATTS do KEYD[K]:=0;
          READ(F,NTPLS);
          GETNAME(ID);
          RELFL:=ID
        end
  end;

procedure GETRELNO(var N:integer;ID:ALFA);
  var
    I:integer;
  begin I:=1;
    while((I<=NORELSP-1)and(RELDIR[I].RELNAMED<>ID)) do I:=I+1;
    if ((I<NORELSP)or((I=NORELSP)and(RELDIR[I].RELNAMED=ID))) then N:=I
    else N:=0
  end;

procedure READRELS;
  var
    I,J,K:integer;
    TPL1,TPL2:TUPLE;
  begin for I:=1 to NORELS do
    with RELDIR[I] do
      begin RESET(F,RELFL);
        NEW(TPL1);
        FRSTPL:=TPL1;
        for K:=1 to NTPLS do
          begin J:=1;
            while ATTD[J]#0 do
              begin
                if ATTDIR[ATTD[J]].TYPEATT=INTNUMBER then
                  READ(F,TPL1^.FIELD[J])
                else
                  begin GETNAME(ID);
                    TPL1^.FIELD[J]:=SEARCH(ATTD[J],ID);
                  end;
                J:=J+1
              end;
            TPL2:=TPL1;
            NEW(TPL1);
            TPL2^.NXTPL:=TPL1
          end;
        TPL2^.NXTPL:=nil
      end
  end;
```

FIGURE 4.8. PROCEDURES DEFINERELS, GETRELNO
AND READRELS

are read by procedure READRELS, a listing of which is given
in Fig. 4.8. Each file contains the values of the attributes
in the same order in which the attribute names were defined
in 'RELDEF'. The integer values are read and stored in the
tuples as such. The non-integer values are read and their
corresponding number is obtained from the tables, which have
been read from the file 'TABLES'. These numbers are stored
in the tuples. A procedure SEARCH, a listing of which is given
in Fig. 4.7, does the job of getting the number for each non .
integer value read.

4.4.3 Common Attribute and Common Relation Tables:

These are the most important tables in the system and
are used at the time of producing the object query. The basic
assumption in our system is made use of in these tables. To
state explicitly, the assumption is that two different rela-
tions in the system are either related through a common attri-
bute, or, if there is no common attribute between the two,
there is atmost one other relation (in the user's sub-schema)
which relates these two relations. In other words, for two
relations R1 and R2, either there is one and only one attribute
A1 which is common to both R1 and R2 (this is true in case the
relations are in 3NF), or there exists atmost one other rela-
tion R3 such that there is an attribute A1 common to both
R1 and R3 and there is another attribute A2 common to both R2
and R3.

In case there is no such Al (common to R1 and R2) and
there is no such R3 (common to R1 and R2) and the source query
involves two such relations, then the object query is not
produced and the user is asked to break up his query into
simpler subqueries so that the above assumption becomes true.
This breaking up is always possible if the query is meaningful.
It has to be pointed out here that in most of the queries,
commonly made by a genuine user, such a condition will not
arise. The provision of giving this message to the user is
required so that he may know that either his subschema is
not complete or that his query is extremely complicated.

Based on this assumption, we build up two tables -COMATT
and COMREL to store all such information. Both these tables
are two dimensional arrays of size NORELSP X NORELSP. The
enteries in these tables are made according to the following
rules --

COMATT [i,j] = 0 if i and j are two relations that have
no attribute in common.

COMATT [i,j] = k if k is an attribute that is common to
relations i and j.

COMREL [i,j] = 0 if there is no relation in common to
relations i and j, or COMATT [i,j] $\neq$ 0.

COMREL [i,j] = k if relation k is common to relations
i and j, and COMATT [i,j] = 0.

Here i,j and k are the sequence numbers of the attributes
and relations in ATTDIR and RELDIR respectively. These tables
are initialized (for the relations read in) by the procedure
BLDCOMTAB. A listing of this procedure is given in Fig.4.9.

In case of a multi-step query, after the processing of
each sub-query, these tables are updated to include the final
relation built by the user in that sub-query. This is done
because the next sub-query is very likely to involve the
relation built by the previous sub-query. A procedure MDFCOMTAB
has been written for the purpose. A listing of this procedure
is given in Fig. 4.10.

4.5 Tables Built (Run Time):

During the processing of any query, information con-
tained in the source query about the type of the condition part
and the attributes and relations involved therein, and the
action required and the attributes and relations involved therein
has to be stored in some form to enable the object query genera-
tor to perform its task. Now we describe these tables in details.

4.5.1 Table for Condition Part:

This has been given the name COMTAB. It is an array (of
size 10) of records for the general case of a complex condition.
This record has four fields common for all types (refer to the
7 condition types described in Chapter III) of conditions and
different number of more fields each depending on the type of
the condition.

```
procedure BLDCOMTABS;
   begin
     for I:=1 to NORELSP do
       for J:=1 to NORELSP do
         begin COMATT[I,J]:=0;
           COMREL[I,J]:=0
         end;
     for I:=1 to NORELS do
       for J:=1 to NORELS do
         if I#J then
           begin FLAG:=false;
             K:=1;
             repeat L:=RELDIR[I].ATTD[K];
                 M:=1;
                 repeat
                   if L=RELDIR[J].ATTD[M] then FLAG:=true;
                   M:=M+1
                 until (FLAG or(RELDIR[J].ATTD[M]=0));
                 K:=K+1
               until (FLAG or(RELDIR[I].ATTD[K]=0));
               if FLAG then COMATT[I,J]:=L
           end;
     for I:=1 to NORELS do
       for J:=1 to NORELS do
         if ((I#J)and(COMATT[I,J]=0)) then
           begin K:=1;
             FLAG:=false;
             repeat
               if ((COMATT[I,K]#0)and(COMATT[J,K]#0)) then FLAG:=true
               else K:=K+1
             until(FLAG or (K>NORELS));
             if FLAG then COMREL[I,J]:=K
           end
   end;

procedure CLEARTABLES;
   var
     I:integer;
     begin I:=1;
       while TABATT[I].RELNOT#0 do
         begin TABATT[I].RELNOT:=0;
           TABATT[I].ATTNOT:=0;
           TABATT[I].ACTCD:=0;
           I:=I+1
         end;
       with ACTTAB do
         begin SRTREQ:=false;
           TTLREQ:=false;
           COUNTREQ:=false;
           UNTREQ:=false
         end
     end;
```

FIGURE 4.9. PROCEDURES BLDCOMTABS
AND CLEARTABLES

```pascal
procedure DEFCOMTAB;
   var
     I,J,K,L:integer;
     FLAG:boolean;
   begin
    for I:=1 to NORELS do
     begin J:=1;
      FLAG:=false;
      repeat
        if TABATT[J].RELNOT=1 then FLAG:=true
        else J:=J+1
      until (FLAG or (TABATT[J].ACTCD=0));
      if FLAG then
       begin L:=J;
        FLAG:=false;
        repeat K:=1;
          repeat
            if TABATT[L].ATTNOT=RELDIR[I].KEYD[K] then FLAG:=true
            else K:=K+1
          until (FLAG or (RELDIR[I].KEYD[K]=0));
          if not FLAG then L:=L+1
        until (FLAG or (TABATT[L].ACTCD=0));
        if FLAG then
         begin COMATT[I,IREL]:=TABATT[L].ATTNOT;
          COMATT[IREL,I]:=TABATT[L].ATTNOT
         end
        else
         begin COMATT[I,IREL]:=TABATT[J].ATTNOT;
          COMATT[IREL,I]:=TABATT[J].ATTNOT
         end
      end
     end;
    J:=1;
    repeat K:=TABATT[J].ATTNOT;
     for I:=1 to NORELS do
      begin
       if COMATT[I,IREL]=0 then
        begin L:=1;
         FLAG:=false;
         repeat
           if RELDIR[I].ATTD[L]=K then FLAG:=true
           else L:=L+1
         until (FLAG or (RELDIR[I].ATTD[L]=0));
         if FLAG then
          begin COMATT[I,IREL]:=K;
           COMATT[IREL,I]:=K
          end
        end
      end;
     J:=J+1
    until TABATT[J].ACTCD=0;
    for I:=1 to NORELS do
     begin
      if COMATT[I,IREL]=0 then
       begin J:=1;
        FLAG:=false;
        repeat
          if ((COMATT[I,J]=0) or (COMATT[IREL,J]=0)) then J:=J+1
          else
           begin COMREL[I,IREL]:=J;
            FLAG:=true;
            COMREL[IREL,I]:=J
           end;
        until ((J>NORELS) or FLAG)
       end
     end
   end;
```

FIGURE 4.10 PROCEDURE DEFCOMTAB

First let us define a variable CNDCD for the condition codes. It is of the type CONDCODE which can take integer values from 1 to 9. Each of these values (except 6 which is not used presently - it can be used for adding any future types of conditions) corresponds to one of the condition types. The four common fields in record CONTAB are -

RELNOC : for storing the relation number of the relatio on the left hand side of the comparison operat in the condition;

ATTNOC : for storing the attribute number of the attribute on the left hand side of the com- parison operator in the condition. It stores the value 'zero' in case CNDCD = 8 or 9. (For CNDCD refer next page);

COND : for storing the comparison operator involved in the condition;

and   CNDCD : this stores the type of the condition.

The variable fields for different values of CNDCD are as follows-

CNDCD = 1 (relation.attribute <COND> " value")

VAL1A
or
VAL1I : to store the value within the quotes, dependin upon whether it is alphanumeric or integer or
or
VAL1R real respectively.

CNDCD = 2 (relation.attribute = *)

No special fields.

First let us define a variable CNDCD for the condition codes. It is of the type CONDCODE which can take integer values from 1 to 9. Each of these values (except 6 which is not used presently – it can be used for adding any future types of conditions) corresponds to one of the condition types. The four common fields in record CONTAB are –

RELNOC : for storing the relation number of the relation on the left hand side of the comparison operator in the condition;

ATTNOC : for storing the attribute number of the attribute on the left hand side of the comparison operator in the condition. It stores the value 'zero' in case CNDCD = 8 or 9. (For CNDCD refer next page);

COND : for storing the comparison operator involved in the condition;

and CNDCD : this stores the type of the condition.

The variable fields for different values of CNDCD are as follows–

CNDCD = 1 (relation.attribute <COND> " value ")

VAL1A
or
VAL1I : to store the value within the quotes, depending upon whether it is alphanumeric or integer or
or
VAL1R   real respectively.

CNDCD = 2 (relation.attribute = *)

No special fields.

CNDCD = 3   (relation.attribute = ALL)

     No special fields.

CNDCD = 4   (relation.attribute <COND> n TOP)

     VAL   · for storing the value of 'n'.

CNDCD = 5   (relation.attribute <COND> n BOTTOM)

     VAL4 . for storing the value of 'n'.

CNDCD = 6   (not used prsently)

CNDCD = 7   (relation.attribute <> relation.attribute)

     RELNO7 : for storing the relation number of the relation
             on the right hand side of the comparison operator.

     ATTNO7 : for storing the attribute number of the attri-
             bute on the right hand side of the comparison
             operator.

CNDCD = 8   (TOTAL(relation SAME attributes:attribute) <COND> value

     SATTS8 : pointer to a list of nodes of the type ATTNOD.
             This list contains the attribute numbers of
             attributes in the SAME list.

     TATT8 : for storing the attribute number of the attri-
             bute of which the total has to be checked.

     VAL8I⎫
      or ⎬ : the value on the right hand side of
     VAL8R⎭   the comparison operator, depending upon whether
             it is integer or real respectively.

CNDCD = 9   (CQUNT(relation SAME attributes DIFFERENT attributes)
         <COND> positive integer)

SATTS9 : pointer to a list of nodes of type ATTNODE.
This list contains the attribute numbers of
attributes in the SAME list.

DATTS9 : pointer to the list of attributes in the
DIFFERENT list.

VAL9 : stores the non-negative integer value on the
right hand side of the comparison operator.

4.5.2 Tables for Action Part:

Two tables are built for the purpose of storing infor-
mation in the action part of the source query. One is used for
storing the numbers of attributes and relations that are required
to perform the task asked for in the action part. Other is
for storing the final action required on the relation to be
outputted to the user's work space. We discuss these tables
in greater detail now.

Table of Attributes Required:

This is named as TABATT. It is an array of size NOATTS.
Each element is a record of type ATTSUSED. It has 3 fields —

ATTNOT : for storing the attribute number of the attribute
in the list of attributes in the action part;

RELNOT : for storing the relation number of the relation
in whose context that attribute has been used;

and ACTCD : (ACTion CoDe) for storing the reason for which
that attribute is used in the action part. This
stores the value 'zero' if the attribute is not

to be outputted in the final relation (and
has to be brought temporarily for the purpose
of sorting and/or totalling). If it is an
output attribute then this field stores the
value 'one'.

In case sorting and/or totalling is required before
finally outputting the relation, the field ACTCD is made use
of to remove those attributes from the final relation which
have 'zero' in this field.

Table for Actions Required:

This is named as ACTTAB. It is a record of the type
ACTION which has the following fields -

SRTREC : a boolean variable which stores 'true' if
          sorting is required before outputting;

PSORT : a pointer to the list of SORTNODES. This
          node contains the attribute number (ATTNOS)
          and sort order (SORDER) of that attribute.
          The first attribute in the list is the major
          field and rest, if any, are subsequent minor
          fields for sorting. SORDER stores 'UP' or
          'DOWN' as the case may be;

TTLREC : a boolean variable which stores 'true' if totall
          ing of a certain domain is required;

TREL   : for storing the relation number of the relation
          involved in doing the totalling;

SATTS : a pointer to the list of attribute numbers
of attributes in the SAME list;

TATT : for storing the number of the attribute of
which the totalling has to be done;

COUNTREQ: a boolean variable which stores 'true' if count
of the number of tuples in the final relation
is required;

LMTREQ : a boolean variable which stores 'true' if the
number of tuples to be outputted is to be
limited to a certain number;

LIMNO : for storing the number of tuples to be out-
putted if LMTREQ is true.

These two tables, ACTTAT and TABATT are cleared before
the processing of any query (or sub-query in case of a multi-
step query). This is done by the procedure CLEARTABLES, a
listing of which is given in Fig. 4.9.

4.6 Object Query Generation.

The runtime tables, that have been just described, and
the system's maintained tables, that were described earlier
in this chapter, are together used to convert the intent of
the source query to statements in the object query. These
statements are the usual relational algebra operations like
project, join, divide etc.

A different sequence of steps is required for each of
the different condition codes (CNDCD) stored in the condition

table (CONTAB). The numbers of the relations involved are stored in CONTAB and TABATT. We denote the relation and the attribute on the left hand side of the comparison operator in the condition part as RC and AC respectively. One idea has to be brought out which is common to all condition codes. It is that TABATT contains the numbers of all relations (along with the attributes) that are to be brought to the work space. Four different cases are possible :

(i)   All the relations in TABATT are same as RC.

(ii)   There exist relations in TABATT which are different from RC but have an attribute each common to it. (This can be checked from the COMATT table described earlier).

(iii)   There exist relations in TABATT which are different from RC, have no common attribute with it, but have a relation each common to it. (This can be checked from the table COMREL described earlier), or

(iv)   None of the above three, i.e., the relations, different from RC, in TABATT do not have even a relation in common with RC.

We will describe, after a while, the sequence of actions for each of the condition codes for case (i) only. However, the system takes care of all cases in the following way –

A procedure CHECK (FLAG,I) checks the relations in TABAT to see if they are different from RC. If it finds no such relation then the value of FLAG is set to 'false' (case (i)).

Else FLAG is set to 'true' and then 'I' points to that entry in TABATT which has such a relation. A procedure BUILD is invoked if FLAG is found to be true. This procedure joins to RC all the relations in TABATT which are different from RC. It scans TABATT completely and for each of the relations found to be different from RC (and any of the previous relations encountered in TABATT so far) it checks up tables COMATT and COMREL. Let R1 be such a relation. If COMATT [R1,RC] is not zero (case (ii)), then it joins to RC (or the relation built up so far as a result of previous such relations encountered in TABATT) this relation R1. If it is zero then it goes on to checking COMREL [R1,RC]. If this is not zero then (case (iii)) R1 and the relation from COMREL are joined to RC (or the relation built so far). If this is also zero (case (iv)) then no actions are taken as it means that the query is either very complicated or it is meaningless. A procedure DUMP is then called which asks the user to split the query into simpler sub-queries and run again as a multi-step query. This is always possible to be done if the query is meaningful. A listing of procedures CHECK, BUILD and DUMP is given in Fig. 4.11.

Now we go on to describing the sequence of relational algebra operations to be performed for each of the 8 types of condition codes for case (i) only.

```
ocedure CHECK(var FLAG:boolean; var I:integer);
  var
    K:integer;
  begin I:=I+1;
    FLAG:=false;
    repeat
      if((TABATT[I].RELNOT#0)and(TABATT[I].RELNOT#RC))then
        begin FLAG:=true;
          if I>1 then
            begin K:=1;
              repeat
                if TABATT[I].RELNOT=TABATT[K].RELNOT then FLAG:=false;
                K:=K+1
              until K>=I
            end
        end;
      if not FLAG then I:=I+1
    until (FLAG or (TABATT[I].RELNOT=0))
  end;

roceaure BUILD;
  procedure DUMP;
    var
      I:integer;
    begin for I:=1 to 4 do WRITELN;
      WRITE('*****ERROR : Split the Query into smaller Queries ');
      WRITELN(' and run again.');
      HALT
    end;
  begin R1:=TABATT[I].RELNOT;
    A1:=COMATT(R1,RC);
    if A1#0 then JOIN(R1,A1,RC)
    else
      begin R2:=COMREL(R1,RC);
        if R2#C then
          begin A1:=COMATT(R1,R2);
            JOIN(R1,A1,R2);
            A2:=COMATT(R2,RC);
            JOIN(RC,A2,IRFL)
          end
        else DUMP
      end;
    CHECK(FLAG,I);
    while FLAG do
      begin R1:=TABATT[I].RELNOT;
        A1:=COMATT(R1,RC);
        if A1#0 then JOIN(R1,A1,IREL)
        else
          begin R2:=COMREL(R1,RC);
            if R2#0 then
              begin A1:=COMATT(R1,R2);
                JOIN(R1,A1,R2);
                A2:=COMATT(RC,R2);
                JOIN(IREL-1,A2,IREL)
              end
            else DUMP
          end;
        CHECK(FLAG,I)
      end
  end;
```

FIGURE 4.11 PROCEDURES CHECK, BUILD
            AND DUMP

<u>CNDCD = 1</u>   <u>(RC.AC <COND> ' value")</u>

In this case RC is projected by the required attribute bringing only those tuples for which 'AC <COND> value' is satisfied.

i.e.,   PROJECT RC BY attributes in TABATT WHERE AC <COND> value;

<u>CNDCD = 2</u>   <u>(RC.AC = *)</u>

This is converted to just the following -

PROJECT RC BY attributes in TABATT;

<u>CNDCD = 3</u>   <u>(RC.AC = ALL)</u>

This is converted to the following three steps -

PROJECT RC BY attributes in TABATT and AC;

PROJECT RC BY AC;

DIVIDE first relation BY second relation;

<u>CNDCD = 4</u>   <u>(RC.AC <COND> n TOP)</u>

This is converted to the following steps -

FIND n-th value of AC from top; and then like CNDCD = 1,

PROJECT RC BY attributes in TABATT WHERE AC <COND> value;

<u>CNDCD = 5</u>   <u>(RC.AC <COND> n BOTTOM)</u>

Here the steps are same as in CNDCD = 4 except that the n-th value from bottom is found.

<u>CNDCD = 7</u>   <u>(RC.AC <> R7.A7)</u>

Here AC is same as A7. This is converted to -

PROJECT RC BY AC; (giving relation 1)

PROJECT R7 BY A7; (giving relation 2)

SUBTRACT relation 2 FROM relation 1; (giving relation 3)

JOIN RC AND relation 3; and finally,

PROJECT above relation by attributes in TABATT;

CNDCD=8 (TOTAL (RC SAME attributes : A8) <COND> value)

This is converted to only one step but the projection involved is more complicated than ordinary projection. The step is —

PROJECT RC BY attributes in TABATT WHERE (total of A8

for the same attributes) <COND> value;

CNDCD=9 (COUNT (RC SAME attributes DIFFERENT attributes)

<COND> integer

Here also the projection operation is slightly complicated. The step is —

PROJECT RC BY attributes in TABATT where/(number of tuples with

the same attributes and the different attributes

<COND> integer;

A procedure RUNPARTQUERY checks the CNDCD for each simple condition in the condition part and produces suitable steps of the object query as explained above. A listing of this procedure is given in Fig. 4.14.

The procedure COND XP which checks the syntax of the complete condition part (which could be an arbitrary expression of simple conditions) builds up a list structure to store the 'infix form' of that part. This list is pointed at by pointer GP1

A procedure is written to convert this infix notation to
'pre-fix notation'. This procedure is named CNVPIFTOPF
and a listing of the same is given in Fig. 4.12. This prefix
form is then used to produce object query statements (operations
UNION and INTERSECTION) by a procedure CMBNREL, a listing of
which is given in Fig. 4.13. This procedure combines all
the final relations (of each simple condition) to produce a
relation which can be outputted after actions, as stored in the
table ACTTAB, have been performed on it.

The four possible actions required to be taken on the
final relation are stored in the table ACTTAB and are -

(i)   finding the total number of tuples in the final relation;

(ii)   doing the total of certain domain in the final relation;

(iii)   sorting the final relation on some domain; and

(iv)   limiting the number of tuples to be retained in the
        final relation.

A flag for each of these is kept in ACTTAB. We note that if the
count of the number of tuples is required then none of the
other three operations are required to be performed. Otherwise
any combination of the other three operations could have been
asked for. The sequence is - totalling, sorting and then
limiting.

A listing of procedure RUNQUERY which performs all
these actions is given in Fig. 4.15 for further details.
Finally procedure OUTREL, a listing of which is also given in
Fig. 4.15, outputs the final relation with proper formating.

```
edure CONTIPTOPP;
ar
SYM:SYMBOL;
begin GP3:=nil;
GP2:=nil;
while GP1#nil do
  begin SYM:=GP1^.GAMASYM;
    if SYM=INTNUMBER then
      begin GP4:=GP1;
        GP1:=GP1^.NEXTGAMA;
        GP4^.NEXTGAMA:=GP3;
        GP3:=GP1
      end
    else
      if SYM=LPAREN then
        begin while GP2^.GAMASYM#RPAREN do
          begin GP4:=GP2;
            GP2:=GP2^.NEXTGAMA;
            GP4^.NEXTGAMA:=GP3;
            GP3:=GP4
          end;
          GP2:=GP2^.NEXTGAMA;
          GP1:=GP1^.NEXTGAMA
        end
      else
        if SYM=RPAREN then
          begin GP4:=GP1;
            GP1:=GP1^.NEXTGAMA;
            GP4^.NEXTGAMA:=GP2;
            GP2:=GP4
          end
        else
          if SYM=ORSYM then
            begin GP4:=GP1;
              GP1:=GP1^.NEXTGAMA;
              GP4^.NEXTGAMA:=GP2;
              GP2:=GP4
            end
          else
            if SYM=ANDSYM then
              begin FLAG:=false;
                repeat
                  if GP2#nil then FLAG:=true
                  else
                    if GP2^.GAMASYM#ORSYM then FLAG:=true
                    else
                      begin GP4:=GP2;
                        GP2:=GP2^.NEXTGAMA;
                        GP4^.NEXTGAMA:=GP3;
                        GP3:=GP2
                      end
                until FLAG;
                GP4:=GP1;
                GP1:=GP1^.NEXTGAMA;
                GP4^.NEXTGAMA:=GP2;
                GP2:=GP4
              end
  end;
  while GP2#nil do
    begin GP4:=GP2;
      GP2:=GP2^.NEXTGAMA;
      GP4^.NEXTGAMA:=GP3;
      GP3:=GP4
    end;
  while GP3#nil do
    begin GP4:=GP3;
      GP3:=GP3^.NEXTGAMA;
      GP4^.NEXTGAMA:=GP1;
      GP1:=GP4
    end
nd;
```

FIGURE 4.19 PROCEDURE CONTIPTOPP

```pascal
procedure CMBNRELS;
   procedure UNION(R1,R2:integer);
   procedure INTERSECT(R1,R2:integer);
   begin GP2:=GP1;
     GP3:=GP2^.NEXTGAMA;
     while GP3#nil do
      begin GP4:=GP3^.NEXTGAMA;
       while GP4^.GAMASYM=INTNUMBER do
        begin GP2:=GP3;
          GP3:=GP4;
          GP4:=GP4^.NEXTGAMA
        end;
       R1:=GP2^.GAMAVAL;
       R2:=GP3^.GAMAVAL;
       if GP4^.GAMASYM=ORSYM then UNION(R1,R2)
       else INTERSECT(R1,R2);
       GP2^.GAMAVAL:=IREL;
       GP2^.NEXTGAMA:=GP4^.NEXTGAMA;
       GP2:=GP1;
       GP3:=GP2^.NEXTGAMA
      end
   end;

   procedure GIVENAME;
     begin
       case (IREL-NORELS) of
          1: RELDIR[IREL].RELNAMED:='WTEMP1   ';
          2: RELDIR[IREL].RELNAMED:='WTEMP2   ';
          3: RELDIR[IREL].RELNAMED:='WTEMP3   ';
          4: RELDIR[IREL].RELNAMED:='WTEMP4   ';
          5: RELDIR[IREL].RELNAMED:='WTEMP5   ';
          6: RELDIR[IREL].RELNAMED:='WTEMP6   ';
          7: RELDIR[IREL].RELNAMED:='WTEMP7   ';
          8: RELDIR[IREL].RELNAMED:='WTEMP8   ';
          9: RELDIR[IREL].RELNAMED:='WTEMP9   ';
         10: RELDIR[IREL].RELNAMED:='WTEMP10  ';
         11: RELDIR[IREL].RELNAMED:='WTEMP11  ';
         12: RELDIR[IREL].RELNAMED:='WTEMP12  ';
         13: RELDIR[IREL].RELNAMED:='WTEMP13  ';
         14: RELDIR[IREL].RELNAMED:='WTEMP14  ';
         15: RELDIR[IREL].RELNAMED:='WTEMP15  ';
         16: RELDIR[IREL].RELNAMED:='WTEMP16  ';
         17: RELDIR[IREL].RELNAMED:='WTEMP17  ';
         18: RELDIR[IREL].RELNAMED:='WTEMP18  ';
         19: RELDIR[IREL].RELNAMED:='WTEMP19  ';
         20: RELDIR[IREL].RELNAMED:='WTEMP20  '
       end
     end;
```

FIGURE 4.13 PROCEDURES CMBNRELS AND GIVENAME

```
procedure RUNPARTQUErv(CN:integer);
    var
    P1,P2,R3,RC:integer;
    A1,A2,AC:integer;
    I,J,L:integer;
    FLAG:boolean;
    PATR1,PATR2:PTR1;
    JP:array[1..NOATTS] of integer;
    procedure FINDNTH(R,A,N:integer;FLAG:boolean);
    procedure CHECK(var FLAG:boolean; var I:integer);
    procedure PROJECT(P:integer;PATR:PTR1);
    procedure JOIN(R1,A,P2:integer);
    procedure BUILD;
    procedure DIVIDE(R1,R2:integer);
    procedure SUBTRACT(R1,R2:integer);
    begin INITELM;
     with CONTAB[CN] do
      begin J:=1;
        NEW(P1);
        PATR1:=P1;
        repeat P11:=P1;
          NEW(P1);
          P11^.ATTNOL:=TABATT[J].ATTNOT;
          P11^.NEXTATT:=P1;
          J:=J+1
        until TABATT[J].ATTNOT=0;
        if CNDCD=3 then
          begin P1^.NEXTATT:=nil;
           P1^.ATTNOL:=ATTNOC
          end
        else P11^.NEXTATT:=nil;
        RC:=RELNOC;
        AC:=ATTNOC;
        case CNDCD of
          1,2,4,5,8,9:
            begin
              if ((CNDCD=4) or (CNDCD=5)) then
                begin
                  if CNDCD=4 then FLAG:=true
                  else FLAG:=false;
                  TOID1:=ATTDIR[AC].TYPEATT;
                  CNDCD:=1;
                  FINDNTH(RC,AC,VAL4,FLAG)
                end;
              I:=0;
              CHECK(FLAG,I);
              if not FLAG then PROJECT(RC,PATR1)
              else
                begin BUILD;
                  PROJECT(IREL,PATR1)
                end;
              with ACTTAB do
              if not(TILREQ or COUNTREQ) then RMVDPLCT(IREL)
            end;
```

FIGURE 4.14 PROCEDURE RUNPARTQUERY   (CONTD)

```
3:
    begin NEW(P1);
     PATR2:=P1;
     P1^.NEXTATT:=nil;
     P1^.ATTNO1:=AC;
     I:=0;
     CHECK(FLAG,I);
     if not FLAG then
       begin PROJECT(PC,PATR1);
        RMVDPLCT(IREL);
        PROJECT(RC,PATR2);
        RMVDPLCT(IREL);
        DIVIDE(IREL-1,IREL)
       end
     else
       begin BUILD;
        PROJECT(IREL,PATR1);
        RMVDPLCT(IREL);
        PROJECT(RC,PATR2);
        RMVDPLCT(IREL);
        DIVIDE(IREL-1,IREL)
       end
    end;
7:
    begin R1:=RELNO7;
     NEW(P1);
     PATR2:=P1;
     P1^.NEXTATT:=nil;
     P1^.ATTFOL:=AC;
     PROJECT(RC,PATR2);
     RMVDPLCT(IREL);
     PROJECT(R1,PATR2);
     RMVDPLCT(IREL);
     SUBTRACT(IREL-1,IREL);
     J:=IREL;
     I:=0;
     CHECK(FLAG,I);
     if not FLAG then
       begin JOIN(RC,AC,J);
        PROJECT(IREL,PATR1)
       end
     else
       begin BUILD;
        JOIN(IREL,AC,J);
        PROJECT(IREL,PATR1)
       end;
     with ACTTAB do
     if not (TTLREQ or COUNTREQ) then RMVDPLCT(IREL)
    end
  end
  end
end;
```

FIGURE 4.14 PROCEDURE RUNQUERY(QUERY)

```
procedure RUNQUERY;
  procedure GIVENAME;
  procedure RUNSDLCT(R:integer);
  procedure RUNPARTQUERY(CN:integer);
  procedure CNVTIFTOPF;
  procedure CMBNRELS;
  procedure FINDCOUNT(B:integer);
  procedure OUTOTAL(R:integer);
  procedure SORTOUT(R:integer);
  procedure LIMITED(R:integer);
  begin
   repeat RUNPARTQUERY(CN);
     GP2:=GP1;
     while GP2^.GAMAVAL#CN do GP2:=GP2^.NEXTGAMA;
     GP2^.GAMAVAL:=IREL;
     CN:=CN-1;
   until CN=0;
   CNVTIFTOPF;
   CMBNRELS;
   with ACTTAB do
    begin
     if COUNTREQ then
      begin
       if(TTLREQ or LMTREQ or SRTREQ) then WARN(2);
       FINDCOUNT(IREL)
      end
     else
      begin
       if TTLREQ then OUTOTAL(IREL);
       if SRTREQ then SORTOUT(IREL);
       if LMTREQ then LIMITED(IREL)
      end
    end;
   if ERR=0 then
    begin WRITELN(WSPACE:AL,' = ',RELDIR[IREL].RELNAMED:AL);
     RELDIR[IREL].RELNAMED:=WSPACE
    end
  end;

procedure OUTREL(R:integer);
  var
   I:integer;
  procedure DISPLAYRFL(R:integer);
  begin
   if ACTTAB.COUNTREQ then
    begin for I:=1 to 60 do WRITE('*');
     for I:=1 to 4 do WRITELN;
     WRITELN('Number of tuples in output relation is ',KOUNT)
    end
   else DISPLAYREL(R);
   for I:=1 to 3 do WRITELN;
   for I:=1 to 60 do WRITE('*');
   WRITELN
  end;
```

FIGURE 4.15 PROCEDURES RUNQUERY AND OUTREL

One more thing to be pointed here is that we have
built a number of intermediate relations during the process
of executing a query. Each of these relations is to be
stored as soon as it is built so that the further operations
can make use of them. 'IREL' is a global variable which
always points to the latest relation built by the system.
Each relation is also given a name by the system. This is
done by a procedure GIVENAME, a listing of which is given
in Fig. 4.13.

In case of a multi-step query, the final relation of
each sub-query is given the name specified as the workspace
area for that subquery. The entire procedure described so far
is repeated for each of the sub-queries that follow. Only
difference is that the COMATT and the COMREL tables are
modified and all runtime action tables are cleared before
starting with each of the subsequent subquery. The last sub-
query ends with a semicolon and then the final relation,
named by the final sub-query, is outputted after proper
formating. A listing of the main program is given in Fig.4.16.

4.7 Summary

In this chapter we have shown how the system converts
the source query (in the newly specified language) to rela-
tional algebra operations (which we have called the object
query). We have described in details the tables and procedures
required for the purpose. Listed in Fig. 4.17 are the outputs

```pascal
const
  NORW=14;
  NOREGSP=24;
  NOATTS=12;
  NORELS=4;
  AL=10;
type
  SYMBOL=(IDENT,INTNUMBER,REALNUMBER,LPAREN,RPAREN,COLON,
          PERIOD,COMMA,SEMICOLON,EOL,LSS,GTR,NEQ,LEQ,GEQ,
          APST,STAR,ACTIONSYM,ALLSYM,BOTTOMSYM,NUL,ORSYM,
          CONDITIONSym,COUNTSYM,SAMESYM,GETSYM,ANDSYM,
          DIFFERENTSym,TOPSYM,TOTALSYM,UPSYM,DOWNSYM);
  ALFA=packed array[1..AL] of char;
  UPDOWN=(UP,DOWN);
  PTR1=^ATTNODE;
  ATTNODE=record
            ATTNOL:1..NOATTS;
            NEXTATT:PTR1
          end;
  SORTNODE=record
             ATTNOS:1..NOATTS;
             SORDER:UPDOWN;
             NEXTSORT:^SORTNODE
           end;
  TUPLE=record
          FIELD:array[1..NOATTS] of integer;
          MARK:boolean;
          NXTPL:^TUPLE
        end;
  PTUPLE=^TUPLE;
  RELATION=record
             RELNAMED:ALFA;
             ATTD:array[1..NOATTS] of 0..NOATTS;
             KEYD:array[1..NOATTS] of 0..NOATTS;
             FRSTPL:^TUPLE;
             NTPLS:integer;
             RELFL:ALFA
           end;
  CONDCODE=1..9;
  TYPEOFID1=IDENT..REALNUMBER;
  ATTRIBUTE=record
              ATTNAMED:ALFA;
              TYPEATT:TYPEOFID1
            end;
  TYPEOFID8=(INTNUMBER..REALNUMBER);
  CONDITION=record
              RELNOC:1..NORELSP;
              ATTNOC:0..NOATTS;
              COND:EOL..GEQ;
              case CNDCD:CONDCODE of
                1:(case TOID1:TYPEOFID1 of
                     IDENT:(VAL1A:ALFA);
                     INTNUMBER:(VAL1I:integer);
                     REALNUMBER:(VAL1R:real));
                4,5,6:(VAL4:integer);
                7:(RELNO7:1..NORELSP;
                   ATTNO7:1..NOATTS);
                8:(SATTS8:PTR1;
                   TATT8:1..NOATTS;
                   case TOID8:TYPEOFID8 of
                     INTNUMBER:(VAL8I:integer);
                     REALNUMBER:(VAL8R:real));
                9:(SATTS9:PTR1;
                   DATTS9:PTR1;
                   VAL9:integer)
            end;
  ATTRUSED=record
             ATTNOT:0..NOATTS;
             RELNOT:0..NORELSP;
             ACTCD:0..40
           end;
```

```
ACTION=record
          SRTREQ:boolean;
          PSORT:^SORTNODE;
          ITLPEQ:boolean;
          TREL:1..NORELSP;
          SATTS:PTR1;
          TALT:1..NOATTS;
          COUNTREQ:boolean;
          LMTREQ:boolean;
          LIMNO:integer
        end;
GAMA=record
          GAMASYM:SYMBOL;
          GAMAVAL:integer;
          NEXTGAMA:^GAMA
        end;
var
  SYM,TEMPSYM:SYMBOL;
  ID,A,ASPACE:ALFA;
  EPR,INTNUM,I,J,K,L,M,N,KOUNT,CN,R1,R2,TOTAL:integer;
  FLAG:boolean;
  P,REALNUM:real;
  KK:0..AL;
  F:TEXT;
  CH,TEMPCH:char;
  CC,LL:0..121;
  WORD:array[1..NORW] of ALFA;
  WSYM:array[1..NORW] of SYMBOL;
  SSYM:array[char] of SYMBOL;
  LINE:array[0..121] of char;
  RELDIR:array[1..NORELSP] of RELATION;
  ATTDIR:array[1..NOATTS] of ATTRIBUTE;
  CONTAB:array[1..5] of CONDITION;
  P1,P11:PTR1;
  ACTTAB:ACTION;
  RELNO:0..NORELSP;
  ATTNO:0..NOATTS;
  TABATT:array[1..NOATTS] of ATTSUSED;
  SP1,SP11:^SORTNODE;
  TEMPSORT:UPDOWN;
  CNMREL:array[1..NORELSP,1..NORELSP] of 0..NORELS;
  CNMATT:array[1..NORELSP,1..NORELSP] of 0..NOATTS;
  IREL:1..NORELSP;
  GP1,GP2,GP3,GP4:^GAMA;
  TPL1,TPL2,TPL3,TPL4:^TUPLE;
  TABLE:array[1..NOATTS] of array[1..10] of ALFA;
procedure INITIALIZE;
PROCEDURE HALT;
PROCEDURE ERROR(N:INTEGER);
PROCEDURE WARN(N:INTEGER);
PROCEDURE GETNAME(ID:ALFA);
PROCEDURE DEFINEATTS;
procedure GETATTNO(var N:integer;ID:ALFA);
procedure READATTS;
function SEARCH(N:integer;ID:ALFA):integer;
procedure DEFINERELS;
procedure GETRELNO(var N:integer;ID:ALFA);
procedure READRELS;
procedure BLDCONTABS;
procedure GETNEXTSYM;
procedure CLEARTABLES;
procedure CHECKQUERY;
procedure RUNQUERY;
procedure MDFCONTAB;
procedure OUTREL(R:integer);
```

FIGURE A.15 MAIN PROGRAM

```
ACTION=record
          SRTREQ:boolean;
          PSORT:^SORTNODE;
          TTLREQ:boolean;
          IREL:1..NORELSP;
          SATTS:PTR1;
          IATT:1..NOATTS;
          COUNTREQ:boolean;
          LNTREQ:boolean;
          LINNO:integer
        end;
 GAMA=record
          GAMASYM:SYMBOL;
          GAMAVAL:integer;
          NEXTGAMA:^GAMA
        end;
var
 SYM,TEMPSYM:SYMBOL;
 ID,A,WSPACE:ALFA;
 ERR,INTNUM,I,J,K,L,M,N,KOUNT,CN,R1,R2,TOTAL:integer;
 FLAG:boolean;
 P,REALNUM:real;
 KK:0..AL;
 F:TEXT;
 CD,TEMPCH:char;
 CC,LL:0..121;
 WORD:array[1..NORW] of ALFA;
 WSYM:array[1..NORW] of SYMBOL;
 SSYM:array[char] of SYMBOL;
 LINE:array[0..121] of char;
 RELDIR:array[1..NORELSP] of RELATION;
 ATTDIR:array[1..NOATTS] of ATTRIBUTE;
 CONTAB:array[1..5] of CONDITION;
 P1,P11:PTR1;
 ACTTAB:ACTION;
 RELNO:0..NORELSP;
 ATTNO:0..NOATTS;
 TABATT:array[1..NOATTS] of ATTSUSED;
 SP1,SP11:^SORTNODE;
 TEMPSORT:UPDOWN;
 COMREL:array[1..NORELSP,1..NORELSR] of 0..NORELS;
 COMATT:array[1..NORELSP,1..NORELSP] of 0..NOATTS;
 IREL:1..NORELSP;
 GP1,GP2,GP3,GP4:^GAMA;
 TPL1,TPL2,TPL3,TPL4:^TUPLE;
 STABLE:array[1..NOATTS] of array[1..10] of ALFA;
 procedure INITIALIZE;
 PROCEDURE HALT;
 PROCEDURE ERROR(N:INTEGER);
 PROCEDURE WARN(N:INTEGER);
 PROCEDURE GETNAME(ID:ALFA);
 PROCEDURE DEFINEATTS;
 procedure GETATTNO(var N:integer;ID:ALFA);
 procedure READATTS;
 function SEARCH(N:integer;ID:ALFA):integer;
 procedure DEFINERELS;
 procedure GETRELNO(var N:integer;ID:ALFA);
 procedure READRELS;
 procedure BLDCONTABS;
 procedure GETNEXTSYM;
 procedure CLEARTABLES;
 procedure CHECKQUERY;
 procedure RUNQUERY;
 procedure MDFCONTAB;
 procedure OUTREL(R:integer);;
```

FIGURE 4.12 MAIN PROGRAM   (CONTD...)

```
begin
  INITIALIZE;
  DEFINEATTS;
  READATTS;
  DEFINERELS;
  READRELS;
  BLDCOMTABS;
  WRITELN;
  GETNEXTSYM;
  while SYM=IDENT do
    begin
      if IREL>NORELS then MDFCOMTAB;
      CLEARTABLES;
      CHECKQUERY;
      WRITELN;
      if ERR=0 then
        begin WRITELN('Query is O.K.:');
          for k:=1 to 60 do WRITE('*');
          WRITELN;
          RUNQUERY
        end
      else WRITELN(ERR:3,' Errors in query.')
    end;
  if SYM=SEMICOLON then WARN(1);
  if ERR=0 then OUTREL(IREL);
  WRITELN
end.
```

FIGURE 4.16 MAIN PROGRAM

of each of the example queries considered in Chapter III
(Query1 through Query9). One more query has been included
to show that the system works satisfactorily for a multi-step
query also. The word statement of this query is -

Query_10 Find the names of those suppliers who supply atleast
         all those jobs that are supplied to by supplier 'Smith'

    In our language, this will be written as -

    QUERY10(W1)
    CONDITION : SUPPLIER.SNAME = " SMITH "
    ACTION · GET SUPPLY.J#
  . QUERY10(W)
    CONDITION : W1.J# = ALL
    ACTION : GET SUPPLIER.SNAME;

The output of this example query is given in Fig. 4.17.

```
QUERY1( )
CONDITION:PART.OOH<"5"
ACTION:GET(3) PART.PNAME,PART.OOH;
```

Query is O.K.:
****

| #TEMP1 | = PROJECT PART | BY PNAME | OOH |

Truncate #TEMP1   keeping only the first 3 tuples.

         = #TEMP1
Display #   with proper formatting.
****

PNAME	OOH
NUT	4
SCREW	3
CAM	2

****

```
QUERY2(W)
CONDITION:PART.OOH=1 TOP
ACTION:GET PART.P#,PART.PNAME;
```

Query is O.K.:
****

Find the   1th highest value of OOH   in PART

| #TEMP1 | = PROJECT PART | BY P# | PNAME |

      = #TEMP1
Display #   with proper formatting.
****

P#	PNAME
4	SCREW

****

FIGURE 4.17 EXAMPLE QUERIES

```
QUERY3(/)
CONDITION:SUPPLY.P#=*
ACTION:GET PART.PNAME,PART.QOH(UP PART.QOH);
```

Query is O.K.:
****

WTEMP1   = JOIN of PART   and SUPPLY   on P#

WTEMP2   = PROJECT WTEMP1   BY PNAME   QOH

Sort WTEMP2   on   ascending QOH

             = WTEMP2
Display W   with proper formatting.
****

PNAME	QOH
COG	1
CAM	2
SCREW	3
NUT	4
BOLT	7
SCREW	9

****


```
QUERY4(*)
CONDITION:COUNT(SUPPLY SAME P# DIFFERENT J#)>2
ACTION:GET(COUNT) SUPPLY.P#;
```

Query is O.K.:
****

WTEMP1   = PROJECT SUPPLY   BY P#
Find the number of tuples in WTEMP1   and store in (KOUNT).

W   = WTEMP1
****

Number of tuples in output relation is   3

****

FIGURE 4.17 EXAMPLE QUERIES   CONT

QUERY5(W)
CONDITION:TOTAL(SUPPLY SAME P#,J#:QTY)>5
ACTION:GET SUPPLY.P#,SUPPLY.J#,TOTAL(SUPPLY SAME P#,J#:QTY);

Query is O.K.
****

WTEMP1   = PROJECT SUPPLY   BY P#   J#   QTY

In WTEMP1   do totalling of QTY   with same P#   J#
W   = WTEMP2
Display w   with proper formatting.
****

P#	J#	QTY
1	4	7
3	1	6
3	5	6
3	7	8

****

QUERY6(W)
CONDITION:JOB.J#=ALL
ACTION:GET SUPPLIER.SNAME,SUPPLIER.LOC;

Query is O.K.
****

WTEMP1   = JOIN of SUPPLIER   and SUPPLY   on S#

WTEMP2   = JOIN of JOB   and WTEMP1   on J#

WTEMP3   = PROJECT WTEMP2   BY SNAME   LOC   J#

WTEMP4   = PROJECT JOB   BY J#

WTEMP5   = DIVIDE WTEMP3   BY WTEMP4

W   = WTEMP5
Display w   with proper formatting.
****

SNAME	LOC
JONES	PARIS

****

FIGURE 4.17 EXAMPLE QUERIES   (CONT)

```
QUERY5(W)
CONDITION:TOTAL(SUPPLY SAME P#,J#:QTY)>5
ACTION:GET SUPPLY.P#,SUPPLY.J#,TOTAL(SUPPLY SAME P#,J#:QTY);
```

Query is O.K..
****

```
WTEMP1      = PROJECT SUPPLY       BY P#           J#          QTY
              -------              --
In WTEMP1     do totalling of QTY        with same P#          J
W           = WTEMP2
Display             with proper formatting.
```
****

P#	J#	QTY
1	4	7
3	1	6
3	5	6
3	7	8

****


```
QUERY6(W)
CONDITION:JOB.J#=ALL
ACTION:GET SUPPLIER.SNAME,SUPPLIER.LOC;
```

Query is O.K..
****

```
WTEMP1      = JOIN of SUPPLIER     and SUPPLY      on S#
              -------                  -----          --

WTEMP2      = JOIN of JOB          and WTEMP1      on J#
              -------                  -----          --

WTEMP3      = PROJECT WTEMP2       BY SNAME     LOC       J#
              --------            --

WTEMP4      = PROJECT JOB          BY J#
              --------            --

WTEMP5      = DIVIDE WTEMP3        BY WTEMP4
              -------            --

            = WTEMP5
Display W            with proper formatting.
```
****

SNAME	LOC
JONES	PARIS

****

FIGURE 4.17 EXAMPLE QUERIES

```
QUE;TITLE:SUPPLIER.S#<>SUPPLY.S#
ACTION:GET SUPPLIER.SNAME,SUPPLIER.LOC;

Query is O.K..
***********************************************************

WTEMP1    = PROJECT SUPPLIER    BY S#
            -------              --
WTEMP2    = PROJECT SUPPLY      BY S#
            -------              --
WTEMP3    = DIFFERENCE OF WTEMP1      AND WTEMP2
            ----------

WTEMP4    = JOIN of SUPPLIER    and WTEMP3      on S#
            ----                               --

WTEMP5    = PROJECT WTEMP4      BY SNAME    LOC
            -------             --
W         = WTEMP5
Display W        with proper formatting.
***********************************************************


          SNAME       LOC
          -----       ---

          CLARK       LONDON



*****************************************************************
```

FIGURE 4.17 EXAMPLE QUERIES   (CONT)

```
QUERY3(s)
CONDITION:SUPPLIER.LOC="ATHENS" AND SUPPLY.J#="4"
ACTION:GET SUPPLIER.SNAME,SUPPLIER.LOC;
```

Query is O.K..
****

WTEMP1   = JOIN of SUPPLIER   and SUPPLY   on S#

WTEMP2   = PROJECT WTEMP1   BY SNAME   LOC

WTEMP3   = PROJECT SUPPLIER   BY SNAME   LOC

WTEMP4   = INTERSECTION OF WTEMP2   and WTEMP3

W   = WTEMP4
Display W   with proper formatting.
****

        SNAME   LOC

        ADAMS   ATHENS

****

FIGURE 4.17 EXAMPLE QUERIES   (CONT'

```
QueryS(w)
CONDITION:SUPPLIER.LOC="ATHENS" AND SUPPLY.J#="4"
ACTION:GET SUPPLIER.SNAME,SUPPLIER.LOC;

Query is O.K..
******************************************************************

WTEMP1      = JOIN of SUPPLIER    and SUPPLY      on S#

WTEMP2      = PROJECT WTEMP1      BY SNAME        LOC

WTEMP3      = PROJECT SUPPLIER    BY SNAME        LOC
WTEMP4      = INTERSECTION OF WTEMP2      and WTEMP3

W           = WTEMP4
Display W            with proper formatting.
******************************************************************


        SNAME       LOC
        -----       ---

        ADAMS       ATHENS


    *********************************************************************
```

FIGURE 4.17 EXAMPLE QUERIES   (CONT

```
QUERY=(W)
CONDITION:PART.COLOR="BLUE" OR PART.COLOR="GREEN"
ACTION:GET SUPPLIER.SNAME,SUPPLIER.LOC;

Query is O.K..
****************************************************************
WTEMP1      = JOIN of SUPPLIER    and SUPPLY     on S#

WTEMP2      = JOIN of PART        and WTEMP1     on P#

WTEMP3      = PROJECT WTEMP2      BY SNAME       LOC

WTEMP4      = JOIN of SUPPLIER    and SUPPLY     on S#

WTEMP5      = JOIN of PART        and WTEMP4     on P#

WTEMP6      = PROJECT WTEMP5      BY SNAME       LOC
WTEMP7      = UNION OF WTEMP3      and WTEMP6

W           = WTEMP7
Display W              with proper formatting.
****************************************************************


            SNAME         LOC

            ADAMS         ATHENS
            JUNES         PARIS
            BLAKE         PAPIS



****************************************************************
```

FIGURE 4.17 EXAMPLE QUERIES

(CONTD)

```
QUERY10(J)
CONDITION:SUPPLIER.SNAME="SMITH"
ACTION:GET SUPPLY.J#
QUERY8(J1)
```

query is O.K.
****

.TEMP1   = JOIN of SUPPLY   and SUPPLIER   on S#
            ----   ---   --

.TEMP2   = PROJECT WTEMP1   BY J#
            -------   --

W   = WTEMP2
```
CONDITION:W.J#=ALL
ACTION:GET SUPPLIER.SNAME,SUPPLIER.LOC;
```

query is O.K.
****

.TEMP3   = JOIN of SUPPLIER   and SUPPLY   on S#
            -------   ---   --

WTEMP4   = JOIN of W   and WTEMP3   on J#
            -------   ---   --

WTEMP5   = PROJECT WTEMP4   BY SNAME   LOC   J#
            -------   --

WTEMP6   = PROJECT W   BY J#
            -------   --

WTEMP7   = DIVIDE WTEMP5   BY WTEMP6
                               --

W1   = WTEMP7
Display.w1   with proper formatting.
****

        SNAME   LOC
        -----   ---

        SMITH   LONDON
        JONES   PARIS

****

FIGURE 4.17 EXAMPLE QUERIES

CHAPTER   V

IMPLEMENTATION OF ALGEBRAIC EXPRESSIONS

So far we have converted the source query into expressions in the relational algebra. These expressions consist of the following operations –

    JOIN
    PROJECT
    DIVIDE
    FIND DIFFERENCE
    SORT
    FIND N-TH VALUE
    COUNT NUMBER OF TUPLES
    DO TOTALLING
AND   TRUNCATE

We now give the algorithms for implementing each of these operations on relations. It has to be pointed out here that, at present, no efficient algorithms exist for carrying out these operations. The efficiency of the existing algorithm depends upon the complexity with which data is stored in the system. A more complex storage, involving the maintenance of lists, pointers, secondary indexes etc., could result in faster algorithms for these operations as the access times would be considerably reduced. But a complex storage structure has the disadvantage of making the task of deleting/updating/ inserting more complicated. For our algorithms we assume that

the relations are stored as a list of tuples in the main memory of the computer. All relations, including the intermediate relations, hence, have to be searched sequentially. The idea of giving these algorithms is just to show that the language we have described produces correct results.

5.1 Join:

This operation is invoked through a call to the procedure JOIN (R1, A, R2), where R1 and R2 are the numbers of the relations to be joined on the common attribute A. The lists of tuples in R1 and R2, pointed at by RELDIR [R1].FRSTPL and RELDIR [R2].FRSTPL respectively, are searched sequentially and whenever there is a match found for the values of attribute A in a tuple in R1 and a tuple in R2 then the two tuples are catenated and written into a new tuple which is added to the list of tuples in the new relation being created. Ofcourse, one of the values of the attribute A is omitted in the process. The value of IREL, the pointer to the latest relation in the system, is increased and RELDIR[IREL].FRSTPL is made to point to the first tuple in the newly created relation. A listing of the procedure JOIN is given in Fig. 5.1.

5.2 Project:

This operation is invoked by a call to the procedure PROJECT(R,PATR), where R is the number of the relation to be projected by the attributes whose numbers are in the list

```pascal
procedure JOIN(R1,A,R2:integer);
    var
    I,J1,J2,K,VAL:integer;
    TPL1,TPL2,TPL3,TPL4:^TUPLE;
    begin IRFL:=IRFL+1;
    GIVENAME;
    WRITE(RELDIR[IREL].RELNAMED:AL,' = JOIN of ');
    WRITE(RELDIR[R1].RELNAMED:AL);
    WRITE(' and ',RELDIR[R2].RELNAMED:AL,' on ');
    WRITELN(ATTDIR[A].ATTNAMED:AL);
    WRITELN('-------':20,'---':15,'--':14);
    WRITELN;
    J1:=1;
    while RELDIR[R1].ATTD[J1]#A do J1:=J1+1;
    J2:=1;
    while RELDIR[R2].ATTD[J2]#A do J2:=J2+1;
    I:=1;
    while RELDIR[R1].ATTD[I]#0 do
     begin RELDIR[IREL].ATTD[I]:=RELDIR[R1].ATTD[I];
      I:=I+1;
     end;
    K:=1;
    while RELDIR[R2].ATTD[K]#0 do
     begin
      if RELDIR[R2].ATTD[K]#A then
       begin RELDIR[IREL].ATTD[I]:=RELDIR[R2].ATTD[K];
        I:=I+1;
        K:=K+1
       end
      else K:=K+1
     end;
    TPL1:=RELDIR[R1].FRSTPL;
    NEW(TPL3);
    RELDIR[IREL].FRSTPL:=TPL3;
    while TPL1#nil do
     begin VAL:=TPL1^.FIELD[J1];
      TPL2:=RELDIR[R2].FRSTPL;
      while TPL2#nil do
       begin
        if TPL2^.FIELD[J2]=VAL then
         begin TPL4:=TPL3;
          NEW(TPL3);
          TPL4^.NXTPL:=TPL3;
          I:=1;
          while RELDIR[R1].ATTD[I]#0 do
           begin TPL4^.FIELD[I]:=TPL1^.FIELD[I];
            I:=I+1
           end;
          K:=1;
          while RELDIR[R2].ATTD[K]#0 do
           begin
            if RELDIR[R2].ATTD[K]#A then
             begin TPL4^.FIELD[I]:=TPL2^.FIELD[K];
              I:=I+1;
              K:=K+1
             end
            else K:=K+1
           end
         end;
        TPL2:=TPL2^.NXTPL
       end;
      TPL1:=TPL1^.NXTPL
     end;
    if RELDIR[IREL].FRSTPL=TPL3 then RELDIR[IREL].FRSTPL:=nil
     else TPL4^.NXTPL:=nil
    end;
```

FIGURE 5.1 PROCEDURE JOIN

pointed at by PATR. As described in 4.6, the projection operation would be of different types depending upon the condition type (CNDCD).

For CNDCD = 1,4 and 5, this operation would be;

PROJECT R BY (attributes pointed at by PATR) WHERE AC <COND> value

For CNDCD = 2,3 and 7, this operation would be :

PROJECT R BY (attributes pointed at by PATR).

For CNDCD = 8, this operation would be :

PROJECT R BY (attributes point d at by PATR) WHERE (total of attribute TATT8 with same SATTS8)<COND> VAL8I.

And for CNDCD = 9, this operation would be :

PROJECT R BY (attributes pointed at by PATR) WHERE (number of tuples with same SATTS9 and different DATTS9)<COND> VAL9.

For all the four types of project operations, the list of tuples, pointed at by RELDIR[R].FRSTPL, is searched sequentially and each time a tuple is found which satisfies the constraint (if any) it is projected out into another tuple, copying only values of those attributes which are there in the list PATR. Each time a new tuple is created, it is appended to the list of tuples pointed at by RELDIR[IREL].FRSTPL, where IREL has been incremented in the beginning of the procedure. A procedure TRNSFR(TPL1,TPL2) is employed to transfer the required attribute values from tuple TPL1 into tuple TPL2. A boolean function STSFD(VAL1,VAL2) checks if

'VAL1 <COND> VAL2' is satisfied. A listing of the procedure PROJECT, procedure TRNSFR and function STsFD is given in Fig. 5.2.

5.3 Division:

This operation is invoked through a call to the procedure DIVIDE(R1,R2), where R1 is the number of the relation which is to be divided by relation whose number is R2. In this case R2 is a unary relation and relation R1 has the attribute of R2 at the end in its list of attributes. Because there is no duplication allowed in a relation, a tuple 'X' will appear in the output list if and only if 'X' is found in R1 as many times as there are tuples in R2. Thus the procedure is similar to that for PROJECT (for CNDCD=9), except that instead of VAL9, this procedure uses KOUNT1, which is equal to the number of tuples in R2. A listing of the procedure DIVIDE is given in Figure 5.3. As usual IREL is incremented in the beginning of the procedure and RELDIR[IREL].FRSTPL points at the list of the tuples in the newly created relation.

5.4 Other Operations:

The remaining operations are relatively simpler and are described below briefly.

5.4.1 Find Difference:

This operation is invoked through a call to the procedure SUBTRACT(R1,R2), where relation R2 is to be subtracted from

```pascal
procedure PROJECT(R:integer;PATR:PTR1);
    var
    I,J,K,J1,JT,TEMP,KOUNT:integer;
    P:PTR1;
    FLAG:boolean;
    TPL1,TPL2,TPL3:^TUPLE;
    JS,JD:array[1..NOATTS] of integer;
    function STSFD(VAL1,VAL2:integer):boolean;
      begin STSFD:=false;
      case CONTAB[CN].COND of
          EQL:
             if VAL1=VAL2 then STSFD:=true;
          NEQ:
             if VAL1#VAL2 then STSFD:=true;
          LEQ:
             if VAL1<=VAL2 then STSFD:=true;
          GEQ:
             if VAL1>=VAL2 then STSFD:=true;
          LSS:
             if VAL1<VAL2 then STSFD:=true;
          GTR:
             if VAL1>VAL2 then STSFD:=true
      end
    end;
    procedure TRNSFR(TPL1,TPL2:PTUPLE);
      var
      i:integer;
      begin I:=1;
        while JP[I]#0 do
          begin TPL2^.FIELD[I]:=TPL1^.FIELD[JP[I]];
          I:=I+1
          end
      end;
    begin IREL:=IREL+1;
    GIVENAME;
    WRITE(RELDIR[IREL].RELNAMED:AL,' = PROJECT ');
    WRITE(RELDIR[R].RELNAMED:AL,' BY ');
    P:=PATR;
    I:=1;
    repeat J:=P^.ATTNO;
        RELDIR[IREL].ATTD[I]:=J;
        WRITE(ATTDIR[J].ATTNAMED:AL,' ');
        K:=1;
        while RELDIR[R].ATTD[K]#J do K:=K+1;
        JP[I]:=K;
        P:=P^.NEXTATT;
        I:=I+1
    until P=nil;
    for K:=I to NOATTS do JP[K]:=0;
    WRITELN;
    WRITELN('--------':20,'--':14);
    with CONTAB[CN] do
    case CONCD of
        1:
            begin TPL1:=RELDIR[R].FRSTPL;
                J1:=1;
                while RELDIR[R].ATTD[J1]#AC do J1:=J1+1;
                NEW(TPL2);
                RELDIR[IREL].FRSTPL:=TPL2;
                while TPL1#nil do
                  begin
                    if STSFD(TPL1^.FIELD[J1],VAL1) then
                    begin TPL3:=TPL2;
                        NEW(TPL2);
                        TPL3^.NXTPL:=TPL2;
                        TRNSFR(TPL1,TPL3)
                    end;
                    TPL1:=TPL1^.NXTPL
                  end;
                if RELDIR[IREL].FRSTPL=TPL2 then RELDIR[IREL].FRSTPL:=nil
                else TPL3^.NXTPL:=nil
            end;
```

FIGURE 5.2 PROCEDURE PROJECT

```
2,3,7:
    begin TPL1:=RELDIR(R).FRSTPL;
     NEW(TPL2);
     RELDIR[IREL].FRSTPL:=TPL2;
     while TPL1#nil do
      begin TPL3:=TPL2;
       NEW(TPL2);
       TPL3^.NXTPL:=TPL2;
       TRNSFR(TPL1,TPL3);
       TPL1:=TPL1^.NXTPL
      end;
     if RELDIR[IREL].FRSTPL=TPL2 then
      RELDIR[IREL].FRSTPL:=nil
     else TPL3^.NXTPL:=nil
    end;
6:
    begin TPL1:=RELDIR[R].FRSTPL;
     I:=0;
     P:=SATTS8;
     while P#nil do
      begin I:=I+1;
       K:=1;
       while RELDIR[R].ATTD[K]#P^.ATTNOL do K:=K+1;
       JS[I]:=K;
       P:=P^.NEXTATT
      end;
     for K:=I+1 to NOATTS do JS[K]:=0;
     JT:=1;
     while RELDIR[R].ATTD[JT]#TATT8 do JT:=JT+1;
     NEW(TPL3);
     RELDIR[IREL].FRSTPL:=TPL3;
     while TPL1#nil do
      begin
       if TPL1^.MARK then
        begin TPL1^.MARK:=false;
         TPL1:=TPL1^.NXTPL
        end
       else
        begin TOTAL:=TPL1^.FIELD[JT];
         TPL2:=TPL1^.NXTPL;
         while TPL2#nil do
          begin
           if not (TPL2^.MARK) then
            begin I:=1;
             K:=JS[I];
             FLAG:=true;
             while (FLAG and (K#0)) do
              begin
               if TPL1^.FIELD[K]#TPL2^.FIELD[K] then
                FLAG:=false
               else
                begin I:=I+1;
                 K:=JS[I]
                end
              end;
             if FLAG then
              begin TOTAL:=TOTAL+TPL2^.FIELD[JT];
               TPL2^.MARK:=true
              end
            end;
           TPL2:=TPL2^.NXTPL
          end;
```

FIGURE 5.2 PROCEDURE PROJECT   (CONTD...)

```
          if STSFD(TOTAL,VAL8I) then
           begin TPL4:=TPL3;
             NEW(TPL3);
             TPL4^.NXTPL:=TPL3;
             TRNSFP(TPL1,TPL4);
             TPL2:=TPL1^.NXTPL;
             while TPL2#nil do
               begin
                 if TPL2^.MARK then
                   begin I:=1;
                     K:=JS[I];
                     FLAG:=true;
                     while (FLAG and(K#0)) do
                       begin
                         if TPL1^.FIELD[K]#TPL2^.FIELD[K] then
                           FLAG:=false
                         else
                           begin I:=I+1;
                             K:=JS[I]
                           end
                       end;
                     if FLAG then
                       begin TPL4:=TPL3;
                         NEW(TPL3);
                         TPL4^.NXTPL:=TPL3;
                         TRNSFR(TPL2,TPL4)
                       end
                   end;
                 TPL2:=TPL2^.NXTPL
               end
             end;
           TPL1:=TPL1^.NXTPL
           end
    end;
    if RELDIR[IREL].FRSTPL=TPL3 then RELDIR[IREL].FRSTPL:=nil
    else TPL4^.NXTPL:=nil
    end;
9:
    begin I:=0;
    P:=SATTS9;
    while P#nil do
      begin I:=I+1;
        K:=1;
        while RELDIR[R].ATTD[K]#P^.ATTNOL do K:=K+1;
        JS[I]:=K;
        P:=P^.NEXTATT
      end;
    for K:=I+1 to NOATTS do JS[K]:=0;
    I:=0;
    P:=DATTS9;
    while P#nil do
      begin I:=I+1;
        K:=1;
        while RELDIR[R].ATTD[K]#P^.ATTNOL do K:=K+1;
        JD[I]:=K;
        P:=P^.NEXTATT
      end;
    for K:=I+1 to NOATTS do JD[K]:=0;
    NEW(TPL3);
    RELDIR[IREL].FRSTPL:=TPL3;
```

FIGURE 5.2 PROCEDURE PROJECT (CONTD.)

```pascal
         TPL1:=RELDIR[R].FRSTPL;
         while TPL1#nil do
          begin
           if TPL1^.MARK then
            begin TPL1^.MARK:=false;
             TPL1:=TPL1^.NXTPL
            end
           else
            begin KOUNT:=1;
             TPL2:=TPL1^.NXTPL;
             while TPL2#nil do
              begin
               if not(TPL2^.MARK) then
                begin I:=1;
                 K:=JS[I];
                 FLAG:=true;
                 while (FLAG and (K#0)) do
                  begin
                   if TPL1^.FIELD[K]#TPL2^.FIELD[K] then
                    FLAG:=false
                   else
                    begin I:=I+1;
                     K:=JS[I]
                    end
                  end;
                 if FLAG then
                  begin I:=1;
                   K:=JD[I];
                   FLAG:=false;
                   while not(FLAG or(K=0)) do
                    begin
                     if TPL1^.FIELD[K]=TPL2^.FIELD[K] then
                      FLAG:=true
                     else
                      begin I:=I+1;
                       K:=JD[I]
                      end
                    end;
                   if not FLAG then
                    begin KOUNT:=KOUNT+1;
                     TPL2^.MARK:=true
                    end
                  end
                end;
               TPL2:=TPL2^.NXTPL
              end;
             if STSFD(KOUNT,VAL9) then
              begin TPL4:=TPL3;
               NEW(TPL3);
               TPL4^.NXTPL:=TPL3;
               TRNSFR(TPL1,TPL4)
              end;
             TPL1:=TPL1^.NXTPL
            end
          end;
         if RELDIR[REL].FRSTPL=TPL3 then RELDIR[IREL].FRSTPL:=nil
         else TPL4^.NXTPL:=nil
        end
   end
 end;
```

FIGURE 5.2 PROCEDURE PROJECT

```
TPL1:=RELDIR[R].FRSTPL;
while TPL1#nil do
  begin
    if TPL1^.MARK then
      begin TPL1^.MARK:=false;
       TPL1:=TPL1^.NXTPL
      end
    else
      begin KOUNT:=1;
       TPL2:=TPL1^.NXTPL;
       while TPL2#nil do
         begin
           if not(TPL2^.MARK) then
             begin I:=1;
              K:=JS[I];
              FLAG:=true;
              while (FLAG and (K#0)) do
                begin
                  if TPL1^.FIELD[K]#TPL2^.FIELD[K] then
                    FLAG:=false
                  else
                    begin I:=I+1;
                     K:=JS[I]
                    end
                end;
              if FLAG then
                begin I:=1;
                 K:=JD[I];
                 FLAG:=false;
                 while not(FLAG or(K=0)) do
                   begin
                     if TPL1^.FIELD[K]=TPL2^.FIELD[K] then
                       FLAG:=true
                     else
                       begin I:=I+1;
                        K:=JD[I]
                       end
                   end;
                 if not FLAG then
                   begin KOUNT:=KOUNT+1;
                    TPL2^.MARK:=true
                   end
                end
             end;
           TPL2:=TPL2^.NXTPL
         end;
       if SYSFD(KOUNT,VAL9) then
         begin TPL4:=TPL3;
          NEW(TPL3);
          TPL4^.NXTPL:=TPL3;
          TRNSFR(TPL1,TPL4)
         end;
       TPL1:=TPL1^.NXTPL
      end
  end;
  if RELDIR[IREL].FRSTPL=TPL1 then RELDIR[IREL].FRSTPL:=nil
  else TPL4^.NXTPL:=nil
end
end
end;
```

FIGURE 5.2 PROCEDURE PROJECT

```
procedure DIVIDE(R1,R2:integer);
  var
    I,KOUNT1,KOUNT2:integer;
    TPL1,TPL2,TPL3,TPL4:^TUPLE;
    FLAG:boolean;
  begin IREL:=IREL+1;
    GIVENAME;
    WRITE(RELDIR[IREL].RELNAMED:AL,' = DIVIDE ');
    WRITE(RELDIR[R1].RELNAMED:AL);
    WRITELN(' BY ',RELDIR[R2].RELNAMED:AL);
    WRITELN('------':19,'--':14);
    WRITELN;
    NEW(TPL3);
    RELDIR[IREL].FRSTPL:=TPL3;
    TPL1:=RELDIR[R2].FRSTPL;
    KOUNT1:=0;
    while TPL1#nil do
      begin KOUNT1:=KOUNT1+1;
        TPL1:=TPL1^.NXTPL
      end;
    I:=1;
    while RELDIR[R1].ATTD[I]#RELDIR[R2].ATTD[I] do
      begin RELDIR[IREL].ATTD[I]:=RELDIR[R1].ATTD[I];
        I:=I+1
      end;
    TPL1:=RELDIR[R1].FRSTPL;
    while TPL1#nil do
      begin
        if TPL1^.MARK then
          begin TPL1^.MARK:=false;
            TPL1:=TPL1^.NXTPL
          end
        else
          begin KOUNT2:=1;
            TPL2:=TPL1^.NXTPL;
            while TPL2#nil do
              begin
                if not(TPL2^.MARK) then
                  begin I:=1;
                    FLAG:=true;
                    while (FLAG and(RELDIR[IREL].ATTD[I]#0)) do
                      begin
                        if TPL1^.FIELD[I]#TPL2^.FIELD[I] then FLAG:=false
                        else I:=I+1
                      end;
                    if FLAG then
                      begin KOUNT2:=KOUNT2+1;
                        TPL2^.MARK:=true
                      end
                  end;
                TPL2:=TPL2^.NXTPL
              end;
            if KOUNT1=KOUNT2 then
              begin TPL4:=TPL3;
                NEW(TPL3);
                TPL4^.NXTPL:=TPL3;
                I:=1;
                while RELDIR[IREL].ATTD[I]#0 do
                  begin TPL4^.FIELD[I]:=TPL1^.FIELD[I];
                    I:=I+1
                  end
              end;
            TPL1:=TPL1^.NXTPL
          end
      end;
    if RELDIR[IREL].FRSTPL=TPL3 then RELDIR[IREL].FRSTPL:=nil
    else TPL4^.NXTPL:=nil
  end;
```

FIGURE 5.2 PROCEDURE DIVIDE

relation R1. Both of these relations are unary and have the same attribute in them. The list of tuples pointed at by RELDIR[R1].FRSTPL is traversed through sequentially. For each of the tuples in the list, the list of tuples pointed at by RELDIR[R2].FRSTPL is searched to see if the tuple in R1 is present in R2. In case a tuple in R1 is present in R2 also then the tuple is removed from the list in R1. Finally IREL is incremented and RELDIR[IREL].FRSTPL is made to point to the first tuple of the remaining tuples in R1. A listing of the procedure SUBTRACT is given in Fig. 5.4.

5.4.2 Find n-th Value:

This operation is invoked through a call to the procedure FINDNTH(R,A,N,FLAG), where the N-th value (from top if FLAG is true and from bottom if FLAG is false) of attribute A in relation R is to be found. The list of tuples in R is traversed through N times and each time the tuple with the maximum (or minimum) value is marked. The tuple which is marked in the end contains the value in which we are interested. This value is copied into the condition table CONTAB[CN].VAL1 (integer or real as the case may be). Finally the list of tuples is tra-, versed once again and all the tuples are unmarked. A listing of the procedure FINDNTH is given in Figure 5.4.

```pascal
procedure SUBTRACT(R1,R2:integer);
    var
    TPL1,TPL2,TPL3:^TUPLE;
    FLAG:boolean;
    begin
    repeat TPL3:=RELDIR[R2].FRSTPL;
      FLAG:=false;
      while not(FLAG or (TPL3=nil)) do
        if TPL3^.FIELD[1]=RELDIR[R1].FRSTPL^.FIELD[1] then
        FLAG:=true
        else TPL3:=TPL3^.NXTPL;
      if FLAG then RELDIR[R1].FRSTPL:=RELDIR[R1].FRSTPL^.NXTPL
    until (not FLAG or (RELDIR[R1].FRSTPL=nil));
    if RELDIR[R1].FRSTPL#nil then
      begin TPL1:=RELDIR[R1].FRSTPL;
        while TPL1#nil do
          begin TPL2:=TPL1^.NXTPL;
          if TPL2#nil then
            begin TPL3:=RELDIR[R2].FRSTPL;
            FLAG:=false;
            while not(FLAG or (TPL3=nil)) do
              if TPL3^.FIELD[1]=TPL2^.FIELD[1] then FLAG:=true
              else TPL3:=TPL3^.NXTPL;
            if FLAG then TPL1^.NXTPL:=TPL2^.NXTPL
            else TPL1:=TPL2
            end
          else TPL1:=TPL2
          end
      end;
    IREL:=IREL+1;
    GIVENAME;
    RELDIR[IREL].ATTD[1]:=RELDIR[R1].ATTD[1];
    RELDIR[IREL].FRSTPL:=RELDIR[R1].FRSTPL;
    WRITE(RELDIR[IREL].RELNAMED:AL,' = DIFFERENCE OF ');
    WRITELN(RELDIR[R1].RELNAMED:AL,' AND ',RELDIR[R2].RELNAMED:AL);
    WRITELN('-----------------':26,'---':15);
    WRITELN
    end;

  procedure FINDNTH(R,A,N:integer;FLAG:boolean);
    var
    TPL1,TPL2:^TUPLE;
    I,J:integer;
    begin WRITE('find the ',N:3,'th value of ');
    WRITELN(ATTDIR[A].ATTNAMED:AL,' in ',RELDIR[R].RELNAMED:AL);
    for I:=1 to 2 do WRITELN;
    J:=1;
    while RELDIR[R].ATTD[J]#A do J:=J+1;
    for I:=1 to N do
      begin TPL1:=RELDIR[R].FRSTPL;
      while TPL1^.MARK do TPL1:=TPL1^.NXTPL;
      TPL2:=TPL1^.NXTPL;
      while TPL2#nil do
        begin
        if not (TPL2^.MARK) then
          case FLAG of
          true:if TPL2^.FIELD[J]>TPL1^.FIELD[J] then TPL1:=TPL2;
          false:if TPL2^.FIELD[J]<TPL1^.FIELD[J] then TPL1:=TPL2
          end;
        TPL2:=TPL2^.NXTPL
        end;
      TPL1^.MARK:=true
      end;
    case ATTDIR[A].TYPEATT of
      IDENT:CONTAB[CN].VAL1I:=TPL1^.FIELD[J];
      INTNUMBER:CONTAB[CN].VAL1I:=TPL1^.FIELD[J];
      REALNUMBER:CONTAB[CN].VAL1R:=TPL1^.FIELD[J]
    end;
    TPL1:=RELDIR[R].FRSTPL;
    while TPL1#nil do
      begin TPL1^.MARK:=false;
      TPL1:=TPL1^.NXTPL
      end
    end;
```

FIGURE 5.4 PROCEDURES SUBTRACT AND FINDNTH

5.4.3 Do Totalling:

This operation is invoked through a call to the procedure DOTOTAL(R), where R is the relation number in which totalling is to be done. The list of the same attributes is pointed at by ACTTAB.SATTS8 and the attribute of which the totalling has to be done is stored in ACTTAB.TATT. The list of tuples in R is traversed and for tuples which have the same values of attributes in the same list, the values of the appropriate field are summed up. IREL is incremented and RELDIR[IREL].FPSTPI is made to point at a new tuple which has the attribute values of attributes in the same list and the total of the attribute ACTTAB.TATT found above. The process is repeated for each new combination of values of same attributes found in R and new tuples are appended to the list in IREL. A listing of the procedure DOTOTAL is given in Fig. 5.5.

5.4.4 Sort:

This operation is invoked through a call to the procedure SORTOUT(R), where R is the relation which is to be sorted as per the information stored in ACTTAB.RSORT. A simple ripple sort methodology is adopted. The procedure sorts the relation only on one major field. A listing of the procedure SORTOUT is given in Fig. 5.6.

5.4.5 Count the Number of Tuples:

This operation is invoked through a call to the proce-dure FINDCOUNT(R), where R is the relation number of which the

```pascal
procedure DOTOTAL(R:integer);
    var
    P:PTR1;
    TPL1,TPL2,TPL3:^TUPLE;
    I,J,K,JT:integer;
    JS:array[1..NOATTS] of integer;
    FLAG:boolean;
    begin IREL:=IREL+1; GIVENAME;
    WRITE('In ',RELDIR[R].RELNAMED:AL,' do totalling of,');
    WRITE(ATTDIR[ACTTAB.TATT].ATTNAMED:AL,' with same ');
    P:=ACTTAB.SATTS;
    while P#nil do
        begin WRITE(ATTDIR[P^.ATTNOL],ATTNAMED:AL,' ');
            P:=P^.NEXTATT
        end;
    WRITELN; I:=1;
    while RELDIR[R].ATTD[I]#0 do
        begin RELDIR[IREL].ATTD[I]:=RELDIR[R].ATTD[I];
            I:=I+1
        end;
    JT:=1;
    while RELDIR[R].ATTD[JT]#ACTTAB.TATT do JT:=JT+1;
    P:=ACTTAB.SATTS; J:=0;
    while P#nil do
        begin J:=J+1; I:=1;
            while RELDIR[R].ATTD[I]#P^.ATTNOL do I:=I+1;
            JS[J]:=I; P:=P^.NEXTATT
        end;
    for I:=J+1 to NOATTS do JS[I]:=0;
    TPL1:=RELDIR[R].FRSTPL;
    if TPL1#nil then
        begin NEW(TPL3); RELDIR[IREL].FRSTPL:=TPL3
        end;
    while TPL1#nil do
        begin
        if TPL1^.MARK then
            begin TPL1^.MARK:=false; TPL1:=TPL1^.NXTPL
            end
        else
            begin TOTAL:=TPL1^.FIELD[JT]; TPL2:=TPL1^.NXTPL;
                while TPL2#nil do
                    begin
                    if not(TPL2^.MARK) then
                        begin I:=1; K:=JS[I]; FLAG:=true;
                            while ((K#0) and FLAG) do
                            begin
                                if TPL1^.FIELD[K]#TPL2^.FIELD[K] then
                                    FLAG:=false
                                else
                                    begin I:=I+1; K:=JS[I]
                                    end
                            end;
                        if FLAG then
                            begin TOTAL:=TOTAL+TPL2^.FIELD[JT];
                                TPL2^.MARK:=true
                            end
                        end;
                    TPL2:=TPL2^.NXTPL
                    end;
                TPL2:=TPL3; NEW(TPL3);
                TPL2^.NXTPL:=TPL3;
                I:=1; K:=JS[I];
                while K#0 do
                    begin TPL2^.FIELD[I]:=TPL1^.FIELD[K];
                        I:=I+1; K:=JS[I]
                    end;
                TPL2^.FIELD[I]:=TOTAL; TPL1:=TPL1^.NXTPL
            end
        end;
    TPL2^.NXTPL:=nil;
    end;
```

FIGURE 5.5 PROCEDURE DOTOTAL

```
procedure SORTOUT(R:integer);
    var
        SP1:^SORTNODE;
        I:integer;
        TPL1,TPL2,TPL3,TPL4:^TUPLE;
        FLIP:boolean;
    begin
        with RELDIR[R] do
        begin WRITE('Sort ',RELNAMED:AL,' on ');
            SP1:=ACTTAB.PSORT;
            while SP1#nil do
            begin
                case SP1^.SORDER of
                    UP  :WRITE(' ascending ',ATTDIR[SP1^.ATTNOS].ATTNAMED:AL);
                    DOWN:WRITE(' descending ',ATTDIR[SP1^.ATTNOS].ATTNAMED:AL)
                end;
                SP1:=SP1^.NEXTSORT
            end;
            WRITELN;
            WRITELN;
            I:=1;
            SP1:=ACTTAB.PSORT;
            while ATTD[I]#SP1^.ATTNOS do I:=I+1;
            if FRSTPL#nil then
            begin TPL1:=FRSTPL^.NXTPL;
                if TPL1#nil then
                repeat FLIP:=false;
                    TPL1:=FRSTPL^.NXTPL;
                    case SP1^.SORDER of
                        UP :
                            if FRSTPL^.FIELD[I]>TPL1^.FIELD[I] then FLIP:=true;
                        DOWN:
                            if FRSTPL^.FIELD[I]<TPL1^.FIELD[I] then FLIP:=true
                    end;
                    if FLIP then
                    begin TPL2:=FRSTPL;
                        FRSTPL:=TPL1;
                        TPL2^.NXTPL:=TPL1^.NXTPL;
                        TPL1^.NXTPL:=TPL2
                    end
                    else
                    begin TPL1:=FRSTPL;
                        TPL2:=TPL1^.NXTPL;
                        TPL3:=TPL2^.NXTPL;
                        if TPL3#nil then
                        begin
                            case SP1^.SORDER of
                                UP:while not((TPL2^.FIELD[I]>TPL3^.FIELD[I])
                                        or(TPL3=nil)) do
                                    begin TPL1:=TPL2;
                                        TPL2:=TPL3;
                                        TPL3:=TPL3^.NXTPL
                                    end;
                                DOWN:while not((TPL2^.FIELD[I]<TPL3^.FIELD[I])
                                        or(TPL3=nil)) do
                                    begin TPL1:=TPL2;
                                        TPL2:=TPL3;
                                        TPL3:=TPL3^.NXTPL
                                    end
                            end;
                            if TPL3#nil then
                            begin TPL4:=TPL2;
                                TPL2:=TPL3;
                                TPL4^.NXTPL:=TPL3^.NXTPL;
                                TPL3^.NXTPL:=TPL4;
                                TPL1^.NXTPL:=TPL2;
                                FLIP:=true
                            end
                        end
                    end
                until not FLIP
            end
        end
    end;
```

FIGURE 8.8 PROCEDURE SORTOUT

number of tuples is to be found. A variable 'KOUNT' is initially set to zero. The list of tuples in R is traversed and for each tuple encountered, the value of KOUNT is incremented by 1. The final value of KOUNT is the required count. A listing of the procedure FINDCOUNT is given in Fig. 5.7.

5.4.6 Truncate:

This operation is invoked through a call to the procedure LIMITED(R), where R is the number of the relation in which the tuples are to be limited to the number stored in ACTTAB.LIMNO. A listing of the procedure LIMITED is given in Fig. 5.7.

5.4.7 Remove Duplicates:

This operation is invoked through a call to the procedure RMVDPLCT(R), where R is the relation from which all the dupli-cate tuples are to be removed. The list of tuples is traversed sequentially and for each tuple, the remaining of the list is searched to find if there exists any more tuples with the same values in them. They are removed from the list if they exist. A listing of procedure RMVDPLCT is given in Fig. 5.7.

5.4.8 Union:

This operation is invoked through a call to the procedure UNION(R1,R2), where R1 and R2 are the two relations to be combined. The two relations have the same attributes. In this the list of tuples in R2 is appended to the list of tuples in R1 and the procedure RMVDPLCT is called to remove the duplicate

```pascal
cedure FINDCOUNT(R:integer);
var
  I:integer;
  TPL1:^TUPLE;
begin TPL1:=RELDIR[R].FRSTPL;
  WRITE('Find the number of tuples in ',RELDIR[R].RELNAMED:AL);
  WRITELN(' and store in (KOUNT).');
  WRITELN;
  I:=0;
  while TPL1<>nil do
    begin I:=I+1;
      TPL1:=TPL1^.NXTPL
    end;
  KOUNT:=I
end;

oceaure LIMITED(R:integer);
var
  I:integer;
  TPL1:^TUPLE;
begin TPL1:=RELDIR[R].FRSTPL;
  WRITE('Truncate ',RELDIR[R].RELNAMED:AL,' keeping only ');
  WRITELN('the first ',ACTTAB.LIMNO:2,' tuples.');
  WRITELN;
  I:=ACTTAB.LIMNO;
  while I>1 do
    begin
      if TPL1<>nil then
        begin TPL1:=TPL1^.NXTPL;
          I:=I-1;
        end
    end;
  TPL1^.NXTPL:=nil
end;

oceaure RMVDUPLCT(R:integer);
var
  I,K:integer;
  FLAG:boolean;
  TPL1,TPL2,TPL3:^TUPLE;
begin TPL1:=RELDIR[R].FRSTPL;
  while TPL1<>nil do
    begin TPL2:=TPL1;
      TPL3:=TPL2^.NXTPL;
      while TPL3<>nil do
        begin I:=1;
          K:=RELDIR[R].ATTD[I];
          FLAG:=true;
          while (FLAG and(K<>0)) do
            begin
              if TPL1^.FIELD[I]<>TPL3^.FIELD[I] then FLAG:=false
              else
                begin I:=I+1;
                  K:=RELDIR[R].ATTD[I]
                end
            end;
          if FLAG then
            begin TPL3:=TPL3^.NXTPL;
              TPL2^.NXTPL:=TPL3
            end
          else
            begin TPL2:=TPL3;
              TPL3:=TPL3^.NXTPL
            end
        end;
      TPL1:=TPL1^.NXTPL
    end
end;
```

FIGURE 5.7 PROCEDURES FINDCOUNT, LIMITED AND RMVDUPLCT

tuples from the resulting list. IREL is incremented and RELDIP[IREL].FRSTPL is made to point to the first tuple of the new list. A listing of the procedure UNION is given in Fig. 5.8.

5.4.9 Intersect:

This operation is invoked through a call to the procedure INTERSECT(R1,(P2), where R1 and R2 are the two relations whose intersection has to be found. The two relations have the same attributes. The list of tuples in R1 is traversed and for each of the tuples, the list R2 is searched to find if the tuple is present in R2 also. If it is, then the tuple is appended to the list of tuples pointed at by RELDIR[IREL].FRSTP where IREL has been incremented in the beginning of the procedure. A listing of the procedure INTERSECT is given in Fig. 5.9.

```
procedure UNION(R1,R2:integer);
  var
    I:integer;
    TPL1:^TUPLE;
  begin IREL:=IREL+1;
    GIVENAME;
    WRITE(RELDIR[IREL].RELNAMED:AL,' = UNION OF ');
    WRITELN(RELDIR[R1].RELNAMED:AL,' and ',RELDIR[R2].RELNAMED:AL);
    WRITELN('----- --':21,'---':15);
    WRITELN;
    I:=1;
    while RELDIR[R1].ATTD[I]#0 do
      begin RELDIR[IREL].ATTD[I]:=RELDIR[R1].ATTD[I];
        I:=I+1
      end;
    TPL1:=RELDIR[R1].FRSTPL;
    if TPL1#nil then
      begin RELDIR[IREL].FRSTPL:=TPL1;
        while TPL1^.NXTPL#nil do TPL1:=TPL1^.NXTPL;
        TPL1^.NXTPL:=RELDIR[R2].FRSTPL;
        RMVDPLCT(IREL)
      end
    else
      begin TPL1:=RELDIR[R2].FRSTPL;
        if TPL1#nil then RELDIR[IREL].FRSTPL:=TPL1
        else RELDIR[IREL].FRSTPL:=nil
      end
  end;
```

```
procedure INTERSECT(R1,R2:integer);
   var
    I:integer;
    FLAG:boolean;
    TPL1,TPL2,TPL3:^TUPLE;
   begin IREL:=IREL+1;
    GIVENAME;
    WRITE(RELDIR[IREL].RELNAMED:AL,' = INTERSECTION OF ');
    WRITELN(RELDIR[R1].RELNAMED:AL,' and ',RELDIR[R2].RELNAMED:AL);
    WRITELN('--------------- --':28,'---':16);
    WRITELN;
    I:=1;
    while RELDIR[R1].ATTD[I]#0 do
     begin RELDIR[IREL].ATTD[I]:=RELDIR[R1].ATTD[I];
      I:=I+1
     end;
    if((RELDIR[R1].FRSTPL=nil)or(RELDIR[R2].FRSTPL=nil)) then
      RELDIR[IREL].FRSTPL:=nil
    else
     begin
      repeat TPL3:=RELDIR[R2].FRSTPL;
        FLAG:=true;
        while(FLAG and(TPL3#nil)) do
         begin I:=1;
          while(FLAG and(RELDIR[R1].ATTD[I]#0)) do
           begin
            if TPL3^.FIELD[I]#RELDIR[R1].FRSTPL^.FIELD[I] then
              FLAG:=false
            else I:=I+1
           end;
          TPL3:=TPL3^.NXTPL
         end;
        if not FLAG then RELDIR[R1].FRSTPL:=RELDIR[R1].FRSTPL^.NXTPL
      until(FLAG or(RELDIR[R1].FRSTPL=nil));
      TPL1:=RELDIR[R1].FRSTPL;
      while TPL1#nil do
       begin TPL2:=TPL1^.NXTPL;
        if TPL2#nil then
         begin TPL3:=RELDIR[R2].FRSTPL;
          FLAG:=true;
          while(FLAG and(TPL3#nil)) do
           begin I:=1;
            while(FLAG and(RELDIR[R1].ATTD[I]#0)) do
             begin
              if TPL1^.FIELD[I]#TPL2^.FIELD[I] then
                FLAG:=false
              else I:=I+1
             end;
            TPL3:=TPL3^.NXTPL
           end;
          if not FLAG then TPL1^.NXTPL:=TPL2^.NXTPL
          else TPL1:=TPL2
         end
        else TPL1:=TPL2
       end;
      RELDIR[IREL].FRSTPL:=RELDIR[R1].FRSTPL
     end
   end;
```

FIGURE

CHAPTER VI

CONCLUSION

6.1 Summary of the Work Done:

In this thesis we have suggested a new query language for information retrieval from a relational data base. We have seen that the language is simple and reasonably complete. The language is based on the concept of decision tables and does not involve more difficult concepts of mapping, recursion etc. It also does not require one to know 'predicate calculus' as does DSL ALPHA.

We have developed a system which processes queries in this language. We have first shown how the system converts statements of the query to relational algebra expressions and then how it applies the algebra operations on the relations in the system. A sample data base was chosen and several types of queries were run on it. The outputs of these have been shown.

The system is general enough in that it can read in any new data base and process queries on that data base. The only change required to be made in the system is that the values of NOATTS, NORELS and NORELSP have to be set to the number of attributes in the data base, the original number of relations in the data base and the total number of relations permitted i

the system respectively. This is done by the CONST declarations in the beginning of the program. Ofcourse, the relations and the attributes of the new data base have to be defined in files RELDEF and ATTDEF respectively.

The system has been extensively tested and,from the satisfactory results it produces, it can be asserted with reasonable confidence that the new language proposed in this thesis is convertible to algebraic operations which can in turn be applied on the relations in the data base.

6.2 Scope for Improvements:

In case of a complex data base, where two relations can be related to each other through more than one relation, the table COMREL could be made three dimensional, storing the list of all such common relations in the third dimension. In that case the user must be asked to define his environment before he states his query. During the process of object query genera- tion, we look up the common number between the entries in the COMREL table and the entries in the array 'ENVIRONMENT'. A query would then look as follows -

```
QUERY(WORKSPACE)
ENVIRONMENT : relation names
CONDITION :  condition part
ACTION : action part;
```

Finally, the algorithms for implementing the algebraic operations, suggested in Chapter V, can be made more efficient by storing the relations as indexed sequential files. In fact, much more work needs to be done to achieve maximum efficiency in the implementation of these basic operations.(Ref. [12]).

REFERENCES

[1]   Date, C.J., 'An Introduction to Data Base Systems',
        Addison-Wesley Publishing Co., 1976.

[2]   Martin, J., 'Principles of Data-Base Management',
        Prentice Hall of India Pvt.Ltd., 1977.

[3]   Weiderhold,G., 'Data Base Design', McGraw Hill Book Co.1977

[4]   Codd, E.F., 'A Data Base Sublanguage Founded on the
        Relational Calculus', Proc. 1971, ACM-SIGFIDET Workshop.

[5]   Codd,E.F., 'Normalized Data Base Structure:A Brief
        Tutorial', Proc. 1971 ACM-SIGFIDET Workshop.

[6]   Codd,E.F., 'A Relational Model of Data for Large Shared
        Data Banks,' CACM, Vol.13,No.6,June 1970, pp. 377-387.

[7]   Astraham,M.M., and Chamberlin,D.D., 'Implementation of
        a Structured English Query Language', CACM, Vol.18,No.10,
        October 1975,

[8]   Goldstein,R.C. and Strnad,A.J.,'The MacAIMS Data Management
        System', P roc. 1970 ACM-SIGFIDET Workshop on Data Des-
        cription and Access.

[9]   Bracchi,G., Fedeli,A., and Paolini,P., 'A Language for
        Relational Data Base Management System', Presented at
        6th Annual Princeton Conference on Information Sciences
        and Systems, Princeton Univ., March 1972.

[10]   Notley,M.G., 'The Peterlee IS/1 System',IBM(UK) Scientific
        Centre Report UKSC-0018, March 1972.

[11]   Chamberlin, D.D. and Boyce,R.F., 'SEQUEL : A Structured
        English Query Language', Proc. 1974 ACM-SIGFIDENT Workshop
        on Data Description, May 1974.

[12]   Smith,J.M. and Philip Yen-Tang Chang,'Optimizing the
        Performance of a Relational Algebra Data Base Interface',
        CACM, Vol.18,No.10, October 1975,

[13]   Codd,E.F., 'Relational Completeness of Data Base
        Sublanguages', In Data Base Systems Courant Computer
        Science Symposium, Vol.6,Prentice Hall (1972).